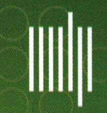
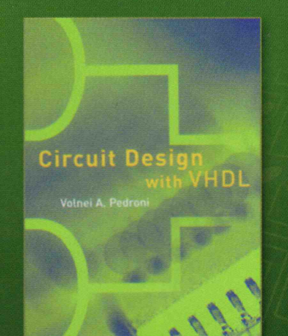
国外电子与通信教材系列



VHDL数字电路设计教程

Circuit Design with VHDL



[巴西] Volnei A. Pedroni 著

乔庐峰 王志功 等译



電子工業出版社
Publishing House of Electronics Industry
http://www.phei.com.cn

VHDL数字电路设计教程

Circuit Design with VHDL

本书采用将数字电路系统设计实例与可编程逻辑相结合的方法，通过大量实例，对如何采用VHDL进行电路设计进行了全面阐述。目前大多数同类教材过多关注于VHDL语法特点本身，而本书则给出了大量完整设计实例的电路图、相关基本概念、电路工作原理以及仿真结果，从而将VHDL语法学习和如何采用它进行电路设计有机地结合起来。本书对VHDL的讲述简明而完整，对与VHDL综合相关的内容进行了详细讨论与说明。全书的内容组织清晰合理，包括电路设计与系统设计两个基本部分，分别讲述了VHDL的基础语法、基本代码编写技术和与VHDL代码分割、共享、重用相关的知识。

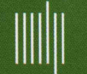
本书的第一部分是电路设计部分，其主要内容包括代码结构、数据类型、操作符和属性、并发和顺序描述语句、对象（信号、变量和常量）、有限状态机的设计以及大量相关例题。第二部分是系统设计部分，讲解了与VHDL电路设计单元库相关的内容，包括包集、元件、函数和过程，同时给出了大量与此相关的例题。附录部分对可编程逻辑器件（PLD/FPGA）的基本结构、发展历史、目前主流厂商及其提供的开发平台的使用进行了详细说明。本书精选了大量典型的设计实例，同时能够将VHDL语法学习和如何采用VHDL进行电路设计有机结合在一起，因此非常适于作为电子工程和计算机科学专业学生的教材。

作者简介

Volnei A. Pedroni：在美国加利福尼亚理工学院获得电子工程博士学位，目前在巴西联邦技术教育中心（CEFET-PR）担任教授。









责任编辑：马　岚
责任美编：毛惠庚

ISBN 7-121-01743-1　　定价：29.00 元

国外电子与通信教材系列

VHDL数字电路设计教程

Circuit Design with VHDL

[巴西] Volnei A. Pedroni　著

乔庐峰　王志功　等译

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内容简介

自从VHDL在1987年成为IEEE标准之后，就因其在电路模型建立、仿真、综合等方面的强大功能而被广泛用于复杂数字逻辑电路的设计中。本书共分为三个基本组成部分，首先详细介绍VHDL语言的背景知识、基本语法结构和VHDL代码的编写方法；然后介绍VHDL电路单元库的结构和使用方法，以及如何将新的设计加入到现有的或自己新建立的单元库中，以便于进行代码的分割、共享和重用；最后介绍PLD和FPGA的发展历史、主流厂商所提供的开发环境的使用方法。本书在内容结构的组织上有独特之处，例如将并发描述语句、顺序描述语句、数据类型与运算操作符和属性等独立成章，使读者更容易清晰准确地掌握这些重要内容。本书注重设计实践，给出了大量完整设计实例的电路图、相关基本概念、电路工作原理以及仿真结果，从而将VHDL语法学习和如何采用它进行电路设计有机地结合在一起。

本书适合通信工程、电子工程及相关专业的高年级本科生作为教材使用，同时也可以作为进行可编程逻辑器件应用开发的培训教材。

版权贸易合同登记号　图字：01-2005-1225

图书在版编目（CIP）数据

VHDL数字电路设计教程/（巴西）佩德罗尼（Pedroni, V. A.）著；乔庐峰等译.
北京：电子工业出版社，2005.9
（国外电子与通信教材系列）
书名原文：Circuit Design with VHDL
ISBN 7-121-01743-1

Ⅰ. V…　Ⅱ. ①佩…　②乔…　Ⅲ. ①硬件描述语言，VHDL-程序设计-教材　②数字电路-电路设计-教材
Ⅳ. ①TP312　②TN79

中国版本图书馆CIP数据核字（2005）第103833号

责任编辑：马　岚　　特约编辑：马爱文
印　　刷：北京市顺义兴华印刷厂
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路173信箱　邮编　100036
开　　本：787×980　1/16　印张：18.75　字数：441千字
印　　次：2009年12月第5次印刷
定　　价：29.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。服务热线：（010）88258888。

序

2001年7月间，电子工业出版社的领导同志邀请各高校十几位通信领域方面的老师，商量引进国外教材问题。与会同志对出版社提出的计划十分赞同，大家认为，这对我国通信事业、特别是对高等院校通信学科的教学工作会很有好处。

教材建设是高校教学建设的主要内容之一。编写、出版一本好的教材，意味着开设了一门好的课程，甚至可能预示着一个崭新学科的诞生。20世纪40年代MIT林肯实验室出版的一套28本雷达丛书，对近代电子学科、特别是对雷达技术的推动作用，就是一个很好的例子。

我国领导部门对教材建设一直非常重视。20世纪80年代，在原教委教材编审委员会的领导下，汇集了高等院校几百位富有教学经验的专家，编写、出版了一大批教材；很多院校还根据学校的特点和需要，陆续编写了大量的讲义和参考书。这些教材对高校的教学工作发挥了极好的作用。近年来，随着教学改革不断深入和科学技术的飞速进步，有的教材内容已比较陈旧、落后，难以适应教学的要求，特别是在电子学和通信技术发展神速、可以讲是日新月异的今天，如何适应这种情况，更是一个必须认真考虑的问题。解决这个问题，除了依靠高校的老师和专家撰写新的符合要求的教科书外，引进和出版一些国外优秀电子与通信教材，尤其是有选择地引进一批英文原版教材，是会有好处的。

一年多来，电子工业出版社为此做了很多工作。他们成立了一个“国外电子与通信教材系列”项目组，选派了富有经验的业务骨干负责有关工作，收集了230余种通信教材和参考书的详细资料，调来了100余种原版教材样书，依靠由20余位专家组成的出版委员会，从中精选了40多种，内容丰富，覆盖了电路理论与应用、信号与系统、数字信号处理、微电子、通信系统、电磁场与微波等方面，既可作为通信专业本科生和研究生的教学用书，也可作为有关专业人员的参考材料。此外，这批教材，有的翻译为中文，还有部分教材直接影印出版，以供教师用英语直接授课。希望这些教材的引进和出版对高校通信教学和教材改革能起一定作用。

在这里，我还要感谢参加工作的各位教授、专家、老师与参加翻译、编辑和出版的同志们。各位专家认真负责、严谨细致、不辞辛劳、不怕琐碎和精益求精的态度，充分体现了中国教育工作者和出版工作者的良好美德。

随着我国经济建设的发展和科学技术的不断进步，对高校教学工作会不断提出新的要求和希望。我想，无论如何，要做好引进国外教材的工作，一定要联系我国的实际。教材和学术专著不同，既要注意科学性、学术性，也要重视可读性，要深入浅出，便于读者自学；引进的教材要适应高校教学改革的需要，针对目前一些教材内容较为陈旧的问题，有目的地引进一些先进的和正在发展中的交叉学科的参考书；要与国内出版的教材相配套，安排好出版英文原版教材和翻译教材的比例。我们努力使这套教材能尽量满足上述要求，希望它们能放在学生们的课桌上，发挥一定的作用。

最后，预祝“国外电子与通信教材系列”项目取得成功，为我国电子与通信教学和通信产业的发展培土施肥。也恳切希望读者能对这些书籍的不足之处、特别是翻译中存在的问题，提出意见和建议，以便再版时更正。

吴佑寿

中国工程院院士、清华大学教授

“国外电子与通信教材系列”出版委员会主任

出 版 说 明

进入21世纪以来，我国信息产业在生产和科研方面都大大加快了发展速度，并已成为国民经济发展的支柱产业之一。但是，与世界上其他信息产业发达的国家相比，我国在技术开发、教育培训等方面都还存在着较大的差距。特别是在加入WTO后的今天，我国信息产业面临着国外竞争对手的严峻挑战。

作为我国信息产业的专业科技出版社，我们始终关注着全球电子信息技术的发展方向，始终把引进国外优秀电子与通信信息技术教材和专业书籍放在我们工作的重要位置上。在2000年至2001年间，我社先后从世界著名出版公司引进出版了40余种教材，形成了一套“国外计算机科学教材系列”，在全国高校以及科研部门中受到了欢迎和好评，得到了计算机领域的广大教师与科研工作者的充分肯定。

引进和出版一些国外优秀电子与通信教材，尤其是有选择地引进一批英文原版教材，将有助于我国信息产业培养具有国际竞争能力的技术人才，也将有助于我国国内在电子与通信教学工作中掌握和跟踪国际发展水平。根据国内信息产业的现状、教育部《关于“十五”期间普通高等教育教材建设与改革的意见》的指示精神以及高等院校老师们反映的各种意见，我们决定引进“国外电子与通信教材系列”，并随后开展了大量准备工作。此次引进的国外电子与通信教材均来自国际著名出版商，其中影印教材约占一半。教材内容涉及的学科方向包括电路理论与应用、信号与系统、数字信号处理、微电子、通信系统、电磁场与微波等，其中既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求，广大师生可自由选择和自由组合使用。我们还将与国外出版商一起，陆续推出一些教材的教学支持资料，为授课教师提供帮助。

此外，“国外电子与通信教材系列”的引进和出版工作得到了教育部高等教育司的大力支持和帮助，其中的部分引进教材已通过“教育部高等学校电子信息科学与工程类专业教学指导委员会”的审核，并得到教育部高等教育司的批准，纳入了“教育部高等教育司推荐——国外优秀信息科学与技术系列教学用书”。

为做好该系列教材的翻译工作，我们聘请了清华大学、北京大学、北京邮电大学、南京邮电大学、东南大学、西安交通大学、天津大学、西安电子科技大学、电子科技大学、中山大学、哈尔滨工业大学、西南交通大学等著名高校的教授和骨干教师参与教材的翻译和审校工作。许多教授在国内电子与通信专业领域享有较高的声望，具有丰富的教学经验，他们的渊博学识从根本上保证了教材的翻译质量和专业学术方面的严格与准确。我们在此对他们的辛勤工作与贡献表示衷心的感谢。此外，对于编辑的选择，我们达到了专业对口；对于从英文原书中发现的错误，我们通过与作者联络、从网上下载勘误表等方式，逐一进行了修订；同时，我们对审校、排版、印制质量进行了严格把关。

今后，我们将进一步加强同各高校教师的密切关系，努力引进更多的国外优秀教材和教学参考书，为我国电子与通信教材达到世界先进水平而努力。由于我们对国内外电子与通信教育的发展仍存在一些认识上的不足，在选题、翻译、出版等方面的工作中还有许多需要改进的地方，恳请广大师生和读者提出批评及建议。

电子工业出版社

教材出版委员会

译 者 序

本书是由 Volnei A. Pedroni 教授编著的一本讲述采用 VHDL 语言进行数字电路设计的教材。我们受电子工业出版社的委托，组织对该书进行了翻译，意在为我国正在蓬勃兴起的集成电路设计人才培养提供可直接使用的教材。

目前国内已有一些 VHDL 语言方面的教材，但这些教材大多数将注意力集中于 VHDL 语法的本身，因此内容显得枯燥和烦琐。在阅读和翻译该书的过程中，我们非常深刻地感觉到该书在内容结构组织、例题选取等方面都与目前国内所有已出版的类似书籍完全不同，具有极其鲜明的特色。本书能够将语法学习和数字电路设计的基本理念通过大量新颖的例题有机地结合起来，从而使读者能够更深刻地理解数字电路设计的基本思想。本书最值得称道的是设置了两个专门的章节集中分析诸如逐级进位加法器和超前进位加法器、定点除法器、乘累加电路、数字滤波器以及神经网络等一系列具有代表意义的电路，使读者既能在学习 VHDL 语言的过程中掌握典型数字电路设计的基本思想和设计技巧，又能在练习设计典型数字电路的过程中掌握 VHDL 语言的应用方法。

在本书的翻译过程中，解放军理工大学通信工程学院的龚坚、彭晖和聂辰分别对第 1 章至第 6 章、第 7 章至第 12 章以及附录进行了初步翻译，乔庐峰对全部译文进行了整理校对，并根据国内教学的具体特点，征得作者同意，对局部内容进行了调整，对一些概念进行了必要的补充和说明，从而使初学者更容易接受。东南大学的王志功教授进一步校对并审核了全书所有章节，提出了大量的修改意见。

鉴于时间紧迫和译者水平，译文中难免有错误之处，敬请读者批评指正。

前　言

本书的结构

全书由电路设计和系统设计两个基本部分组成，其中电路设计部分详细介绍了 VHDL 语言的背景知识、基本语法结构和 VHDL 代码的编写方法。这一部分主要包括以下章节：

- 代码结构：库、实体和构造体（第 2 章）
- 数据类型（第 3 章）
- 运算操作符和属性（第 4 章）
- 并发描述语句和并发代码（第 5 章）
- 顺序描述语句和顺序代码（第 6 章）
- 对象：信号、变量和常量（第 7 章）
- 有限状态机的设计（第 8 章）
- 简单电路设计实例分析（第 9 章）

系统设计部分主要介绍了 VHDL 电路单元库的结构和使用方法，以及如何将新的设计加入到现有的或自己新建立的单元库中，以便于进行代码的分割、共享和重用。这一部分主要包括以下章节：

- 包集和元件（第 10 章）
- 函数和过程（第 11 章）
- 系统设计实例分析（第 12 章）

本书的特色

本书具有以下鲜明特色：

- 用简明的语言介绍了与 VHDL 语言综合相关的所有关键特征。
- 全书的内容结构组织合理，顺序安排得当。全书在内容组织上划分为电路级设计和系统级设计两个基本部分。在电路级设计部分，第 1 章至第 4 章介绍了 VHDL 语言的基本语法知识；第 5 章至第 9 章分析了 VHDL 语言的并发代码、顺序代码、信号与变量、状态机等内容，并给出了大量的设计实例。在系统级设计部分，第 10 章至第 12 章从系统的角度讨论了包集、元件、函数和过程，并分析了许多系统级电路设计的例子。
- 尽量用关联紧密的内容来组织每一章。例如，并发描述语句、顺序描述语句、数据类型以

及运算操作符和属性都独立成章。

- 目前常见的 VHDL 语法书籍往往忽视了数字电路设计的基本概念，而关于数字电路设计的书籍又对 VHDL 介绍得很简单。本书很好地将二者结合起来，通过大量数字电路设计实例来学习 VHDL 语言。
- 为了将 VHDL 语言和数字电路设计实践紧密地结合起来，本书采用了以下方法：
  ➢提供了大量的完整设计实例（而不是片段）；
  ➢对设计实例的顶层电路图进行了解释说明；
  ➢对数字电路设计的重要基本概念进行了回顾；
  ➢对问题的解决方法给出了详细的注释；
  ➢所有设计实例都是可综合、可物理实现的（可以使用可编程逻辑器件实现）；
  ➢提供了仿真结果，并对仿真结果进行了必要的分析和说明；
  ➢最后，在附录中介绍了目前主流的可编程器件以及相应开发工具的使用。

本书适用对象

本书可以作为电子工程和计算机科学专业开设的以下课程的教材：

- VHDL 语言
- 数字电路设计自动化
- 可编程逻辑器件
- 数字电路设计（基础课程和高级课程）

另外，本书也适合作为企业内部 VHDL 培训和可编程逻辑器件应用开发的培训教材。

致谢

首先对所有给本书提出了宝贵意见和建议的人表示感谢。Ricardo P. Jasinski 和 Bruno U. Pedroni 评阅了本书，在此特别表示感谢。

目　　录

第一部分　电 路 设 计

第二部分 系 统 设 计

第一部分　电路设计

第1章 引 言

1.1 关于 VHDL

VHDL 是一种硬件描述语言，它可以对电子电路和系统的行为进行描述。基于这种描述，结合相关的软件工具，可以得到所期望的实际电路与系统。

VHDL 的含义是 VHSIC Hardware Description Language（VHSIC 硬件描述语言）。VHSIC 是 Very High Speed Integrated Circuits 的缩写，是 20 世纪 80 年代在美国国防部的资助下始创的，并最终导致了 VHDL 语言的出现。它的第一个规范版本为 VHDL 87，VHDL 93 是其后续的升级版本。VHDL 是 IEEE（Institute of Electrical and Electronics Engineers，美国电气和电子工程师协会）制定为规范的第一种硬件描述语言，规范版本为 IEEE 1076。IEEE 后来又补充制定了 IEEE 1164，引入了多值逻辑系统。

使用 VHDL 语言描述的电路，可以进行综合与仿真。然而，值得注意的是，尽管所有 VHDL 代码都是可仿真的，但并不是所有代码都是可综合的。

VHDL（或其竞争者 Verilog HDL 语言）被广泛使用的基本原因在于它是一种标准语言，是与工具和工艺无关的，从而可以方便地进行移植和重用。VHDL 语言的两个最直接的应用领域是可编程逻辑器件和专用集成电路（ASIC：Application Specific Integrated Circuits），其中可编程逻辑器件包括复杂可编程逻辑器件（CPLD：Complex Programmable Logic Devices）和现场可编程门阵列（FPGA：Field Programmable Gate Arrays）。一段 VHDL 代码编写完成后，用户可以使用 Altera，Xilinx 或 Atmel 等厂商的可编程器件来实现整个电路，或者将其提交给专业的代客户加工的工厂用于 ASIC 的生产，这也是目前许多复杂的商用芯片（例如微控制器）所采用的实现方法。

关于 VHDL 语言，最后需要说明的是：与常规的顺序执行的计算机程序不同，VHDL 从根本上讲是并发执行的。因此，我们通常称之为代码，而不是程序。在 VHDL 中，只有在进程（PROCESS）、函数（FUNCTION）和过程（PROCEDURE）内部的语句才是顺序执行的。

1.2 设计流程

如上一节所述，使用 VHDL 语言的主要原因之一是通过代码综合，可以采用可编程器件（PLD 或 FPGA）或 ASIC 来实现所需的电路。图 1.1 给出了采用 VHDL 进行设计综合的流程。如图 1.1 所示，设计的第一个阶段是编写 VHDL 代码，编写后的代码保存为一个后缀名为.vhd 的文件（注意，文件名和代码中的实体名应保持一致）。代码编写完毕后进入综合阶段。综合阶段的

第一步是进行代码编译。代码编译过程把寄存器传输级（RTL：Register Transfer Level）的 VHDL 代码转换成门级网表。综合阶段的第二步是优化，主要是根据对电路工作速度和占用硬件资源大小等的要求，对门级网表进行优化。在综合阶段，可以对设计进行仿真。仿真通过后，布局布线工具可以在具体的 PLD/FPGA 器件上对各种电路单元进行布局布线工作，或者生成 ASIC 掩膜文件，用于 ASIC 的生产。

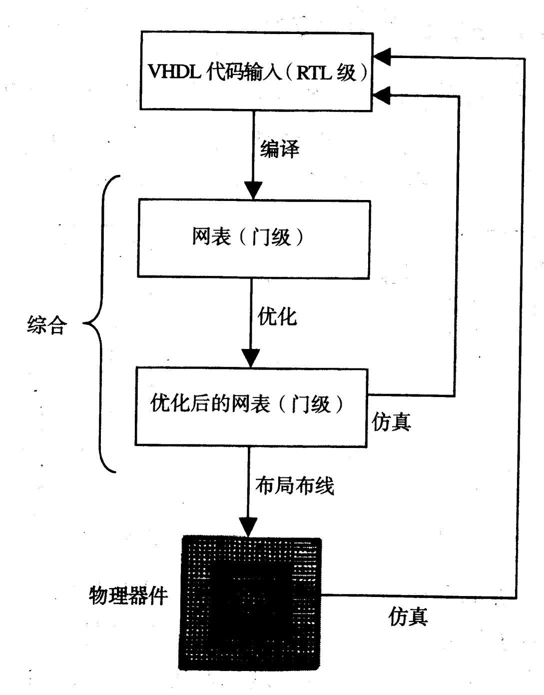


图 1.1　VHDL 设计流程

1.3　EDA 工具

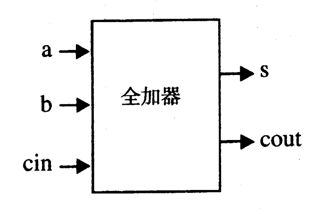
目前有多种 EDA（Electronic Design Automation，电子设计自动化）工具支持采用 VHDL 进行电路综合、仿真以及实现。一些可编程器件生产厂商将使用 VHDL 进行电路设计所需的多种 EDA 工具集成为统一的开发平台提供给用户，进行针对本公司可编程器件产品的开发，从而使整个设计流程更加简捷和易于使用。目前比较常见的是 Altera 公司的 Quartus II 开发平台和 Xilinx 公司的 ISE 开发平台。这些平台中使用的综合工具和仿真工具通常由专业的 EDA 厂商提供，而这些 EDA 厂商除了为这些开发平台提供量身定制的工具外，还推出了具有标准接口的专业设计工具。如 Mentor Graphics 公司的 Leonardo Spectrum（综合工具），Synopsis 公司的 Design Compiler（综合工具），Synplicity 公司的 Synplify（综合工具）以及 Model Technology 公司的 ModelSim（仿真工具）等。

本书提供的设计实例都可以在 Altera 或 Xilinx 的 CPLD/FPGA（见附录 A）设计平台上实现，部分例题的综合是使用 Leonardo Spectrum 完成的。

尽管在本书例题的实现和仿真中使用了不同种类的 EDA 工具，但我们还是尽力使电路的仿真波形在视觉效果上达到统一。由于具有简单清晰的界面，对于大多数设计，使用了 MaxPlus II（见附录 C）的波形编辑器。由于一些较新的开发平台（如 Xilinx 的 ISE 和 Altera 的 Quartus II）所提供的仿真器具备更强大的定时分析功能，所以在对例题的某些细节进行分析时使用了这些工具。

1.4 从 VHDL 代码到电路的转化

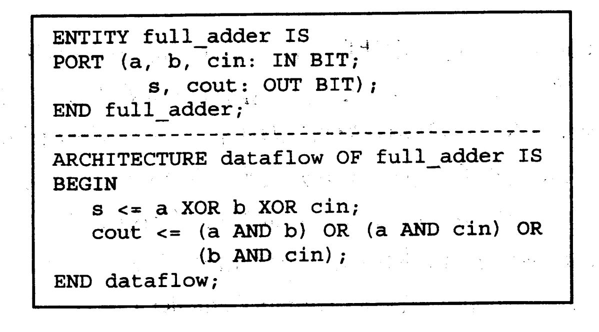
图 1.2 是一个全加器的电路图。图中 a 和 b 是要输入相加的两个位，cin 是输入的进位位，s 是求和结果，cout 是输出的进位位。如图中的真值表所示，当输入端出现奇数个高电平时，s 输出高电平，当输入端出现两个或两个以上高电平时，cout 输出高电平。



a b	cin	s	cout
0 0	0	0	0
0 1	0	1	0
1 0	0	1	0
1 1	0	0	1
0 0	1	1	0
0 1	1	0	1
1 0	1	0	1
1 1	1	1	1

图 1.2 全加器电路框图和真值表

图 1.3 给出了图 1.2 所示全加器的 VHDL 代码。这些代码由实体（ENTITY）和构造体（ARCHITECTURE）两部分组成，其中 ENTITY 给出了电路外部连接端口（PORTS）的定义。ARCHITECTURE 内部的语句描述了电路所实现的功能。从 ARCHITECTURE 中可以看出加法运算的结果 s 是 a，b 和 cin 进行“异或”操作的结果，而进位输出 cout 的运算表达式为 cout = a.b + a.cin + b.cin。



```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
        s, cout: OUT BIT);
END full_adder;
--------------------------------------
ARCHITECTURE dataflow OF full_adder IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin) OR
           (b AND cin);
END dataflow;
```

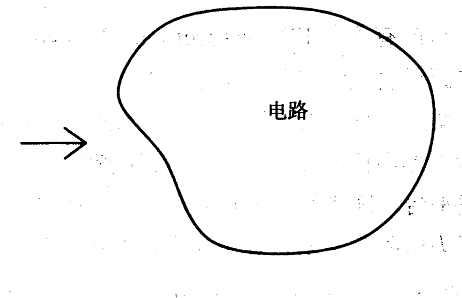


图 1.3 与图 1.2 所示电路对应的 VHDL 代码

根据图 1.3 左侧的 VHDL 代码，可以对应地生成一个实现相同功能的具体电路。根据数字电

路设计的相关知识，有多种具体的电路结构可以实现 ARCHITECTURE 中表达式所描述的功能。具体采取哪种电路结构来实现，取决于所选用的 VHDL 编译器类型、电路优化的目标（希望得到的电路拥有更高的速度还是占用更少的逻辑资源），更重要的是取决于最终所采用的实现方式（使用 PLD 还是 FPGA 来实现）。假如我们使用可编程逻辑器件来实现该电路，可以采用图 1.4（b）和图 1.4（c）给出的两种可能的电路结构。这是满足设计要求的许多结构中的两种，当然这两个电路都满足 cout = a.b + a.cin + b.cin。另一方面，如果希望采用 ASIC 来实现，可以采用图 1.4（d）给出的一种晶体管级的电路结构（采用 MOS 晶体管和时钟多米诺逻辑）。此外，综合工具可以根据需要选择对速度还是对面积进行优化，这也将明显地影响最终的电路结构。

无论代码综合后的电路结构最终是什么样的，如图 1.1 所示，必须在综合完成后对电路的功能进行验证。当然，在实现了物理电路后仍然需要对电路的功能进行测试和验证，但是如果此时对物理电路做出改动，则需要付出很大的代价。

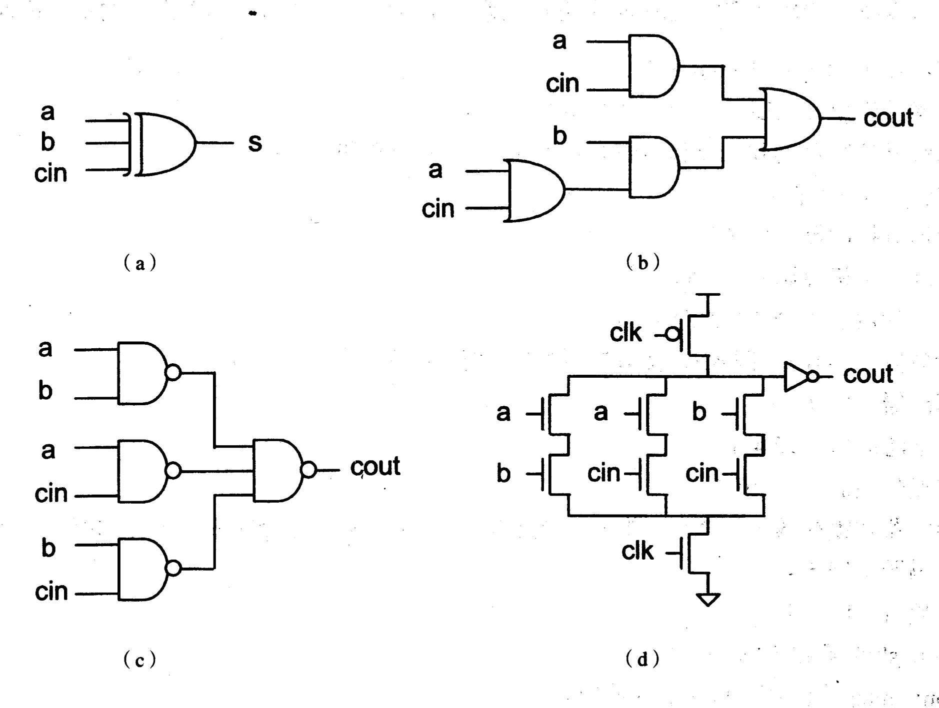


图 1.4 根据图 1.3 所示的全加器 VHDL 代码得到的部分电路结构

图 1.5 给出了图 1.2 中全加器的 VHDL 代码综合后的仿真波形。我们可以看出，输入引脚（用一个内部嵌有标志“I”的向内的箭头表示）和输出引脚（用一个内部嵌有标志“O”的向外的箭头表示）就是在图 1.3 中 ENTITY 后面列出的引脚。我们可以随意地设置输入信号（如本例中的 a，b 和 cin）的值，仿真器根据输入信号的值计算出对应的输出结果（s 和 cout），并将其显示在波形编辑器中。可以看到，图 1.5 中输出了设计期望的波形。

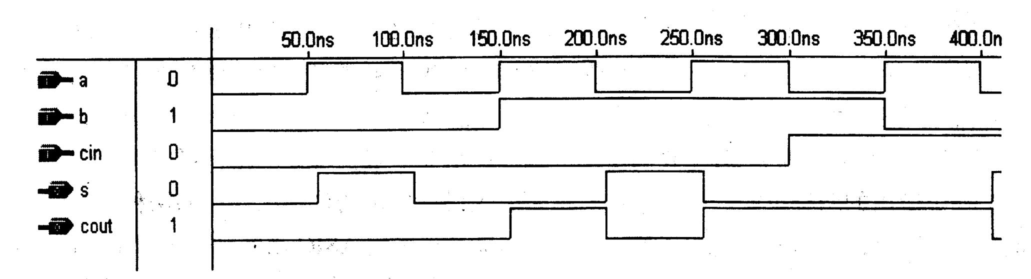


图 1.5 图 1.3 所示代码的仿真结果

1.5 设计实例

如前言所述，本书是通过对大量例题的分析来学习 VHDL 的。通过详细讨论一系列典型的设计实例，可以将 VHDL 和数字电路本身进行良好的结合。下面列出了本书提供的所有设计实例：

- 加法器（例 3.3、例 6.8 和 9.3 节）
- 算术逻辑单元 ALU（例 5.5 和例 6.10）
- 桶形移位寄存器和矢量移位寄存器（例 5.6、例 6.9 和 9.1 节）
- 比较器（9.2 节）
- 交通灯控制器（例 8.5）
- 自动售货机控制器（9.5 节）
- “1” 计数器（例 7.1 和例 7.2）
- 计数器（例 6.2、例 6.5、例 6.7、例 7.7 和例 8.1）
- 解码器（例 4.1）
- 数字滤波器（12.4 节）
- 定点分频器（9.4 节）
- 触发器和锁存器（例 2.1、例 5.7、例 5.8、例 6.1、例 6.4、例 6.6、例 7.4 和例 7.6）
- 编码器（例 5.4）
- 分频器（例 7.5）
- arith_shift 函数（例 11.7）
- conv_integer 函数（例 11.2 和例 11.5）
- 乘法器函数（例 11.8）
- 扩展的 “+” 函数（例 11.6）
- 上升沿（positive_edge）函数（例 11.1、例 11.3 和例 11.4）
- 连零检测器（例 6.10）
- 多路选择（复用）器（例 5.1、例 5.2 和例 7.3）
- 乘法器（例 11.8、12.1 节和 12.2 节）

- MAC 电路（12.3 节）
- 神经网络（12.5 节）
- 并–串转换器（9.7 节）
- 奇偶校验产生电路（例 4.2）
- 奇偶校验检测电路（例 4.3）
- 7 段数码显示器驱动电路（9.8 节）
- min_max 过程（例 11.9 和例 11.10）
- RAM（例 6.11 和 9.10 节）
- ROM（9.10 节）
- 串行数据接收器（9.6 节）
- 移位寄存器（例 6.3、例 7.8 和例 7.9）
- 信号产生器（例 8.6 和 9.9 节）
- 字符串检测器（例 8.4）
- 三态缓冲电路/总线（例 5.3）

此外，在每章之后的习题中还涉及到以下电路：

- 加法器和减法器（习题 3.5、习题 5.4、习题 5.5、习题 6.14、习题 6.16、习题 10.2 和习题 10.3）
- 算术逻辑单元（习题 6.13 和习题 10.1）
- 桶形移位寄存器和矢量移位寄存器（习题 5.7、习题 6.12、习题 9.1 和习题 12.2）
- 二进制码–格雷码转换器（习题 5.6）
- 比较器（习题 5.8 和习题 6.15）
- “1”计数器（习题 6.9）
- 计数器（习题 7.5 和习题 11.6）
- 数据延迟电路（习题 7.2）
- 解码器（习题 4.4 和习题 7.6）
- D 触发器（DFF）（习题 6.17、习题 7.3、习题 7.4 和习题 7.7）
- 数字 FIR 滤波器（习题 12.4）
- 除法器（习题 5.3 和习题 9.2）
- 边沿计数器（习题 6.1）
- 有限状态机（习题 8.1）
- 通用分频器（习题 6.4）
- 倍频器（习题 6.5）
- conv_std_logic_vector 函数（习题 11.1）
- 整型“not”函数（习题 11.2）
- 整型移位操作函数（习题 11.4）

- std_logic_vector 型移位操作函数（习题 11.3）
- BCD-SSD 转换函数（习题 11.6）
- std_logic_vector 型的“+”函数（习题 11.8）
- 密度编码器（习题 6.10）
- 按键防抖动电路和编码器（习题 8.4）
- 多路选择（复用）器（习题 2.1、习题 5.1 和习题 6.11）
- 乘法器（习题 5.3、习题 11.5 和习题 12.1）
- 乘-累加电路（习题 12.3）
- 神经网络（习题 12.5）
- 奇偶校验检测器（习题 6.8）
- 7 段数码显示器（SSD）驱动电路（习题 9.6）
- 优先级编码器（习题 5.2 和习题 6.3）
- 数值统计电路（习题 11.7）
- 随机数产生器与 SSD（习题 9.8）
- ROM（习题 3.4）
- 串行数据接收器（习题 9.4）
- 串行数据发送器（习题 9.5）
- 移位寄存器（习题 6.2）
- 信号产生器（习题 8.2、习题 8.3、习题 8.6 和习题 8.7）
- 速度监视器（习题 9.7）
- 跑表（习题 10.4）
- 计时器（习题 6.6 和习题 6.7）
- 交通灯控制器（习题 8.5）
- 自动售货机控制器（习题 9.3）

在 4 个附录中介绍了以下可编程逻辑器件和综合工具：

- 附录 A：可编程逻辑器件
- 附录 B：Xilinx ISE 与 ModelSim 自学教程
- 附录 C：Altera MaxPlus II 与 Advanced Synthesis Software 使用指南
- 附录 D：Altera Quartus II 使用指南

第 2 章　VHDL 代码结构

本章将分析构成一段完整 VHDL 代码的 3 个基本组成部分：库（LIBRARY）声明、实体（ENTITY）和构造体（ARCHITECTURE）。

2.1　VHDL 代码基本单元

如图 2.1 所示，一段独立的 VHDL 代码至少包含 3 个组成部分：

- 库（LIBRARY）声明：列出了当前设计中需要用到的所有库文件，如 ieee，std 和 work 等。
- 实体（ENTITY）：定义了电路的输入/输出引脚。
- 构造体（ARCHITECTURE）：所包含的代码描述了电路要实现的功能。

库是一些常用代码的集合，将电路设计中经常使用的一些代码存放到库中有利于设计的重用和代码共享，图 2.2 给出了库的典型结构。代码通常以函数（FUNCTION）、过程（PROCEDURE）或元件（COMPONENT）等标准形式存放在包集（PACKAGE）中，用户可以根据需要对其进行编译使用。

本书的第一部分（第 1 章到第 9 章）介绍 VHDL 的基本语法知识（见图 2.1），与库相关的内容（见图 2.2）将在第二部分（第 10 章到第 12 章）中进行讨论。

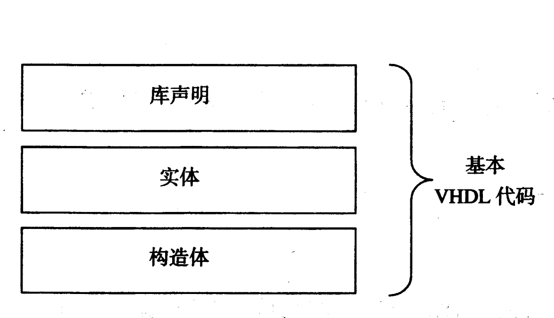


图 2.1　一段 VHDL 代码的基本组成部分

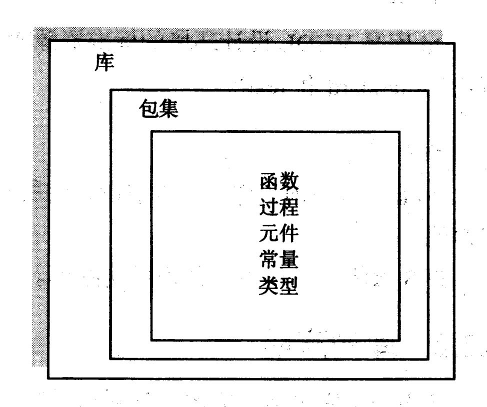


图 2.2　一个库的基本组成部分

2.2 库声明

库（LIBRARY）的建立和使用有利于设计重用和代码共享，同时可以使代码结构更加清晰。

2.2.1 库的种类

在 VHDL 设计中有 3 个常用的库：ieee 库，std 库和 work 库。

ieee 库

在 ieee 库中有一个 IEEE 正式认可的标准包集 std_logic_1164.all。实际上，ieee 库包含了许多包集，列举如下：

- std_logic_1164：定义了 STD_LOGIC（8 值）和 STD_ULOGIC（9 值）多值逻辑系统。
- std_logic_arith：定义了 SIGNED（有符号）和 UNSIGNED（无符号）数据类型和相关的算术运算和比较运算操作。它包含许多数据类型转换函数，这种函数可以实现数据类型的转换。常用的数据类型转换函数包括 conv_integer(p)，conv_unsigned(p, b)，conv_signed(p, b)和 conv_std_logic_vector(p, b)。
- std_logic_signed：内部包含一些函数，这些函数可以使 STD_LOGIC_VECTOR 类型的数据像 SIGNED 类型的数据一样进行运算操作。
- std_logic_unsigned：内部包含一些函数，这些函数可以使 STD_LOGIC_VECTOR 类型的数据像 UNSIGNED 类型的数据一样进行操作。

std 库

std 库是 VHDL 设计环境的标准资源库，包括数据类型和输入/输出文本等内容。std 库中存放有包集 standard 和 textio。

work 库

work 库是当前工作库，当前设计的所有代码都存放在 work 库中，使用 work 库不需要进行任何声明。

第 3 章将进一步讨论并使用这些库。

2.2.2 库的声明

使用一个库之前，需要首先对库进行声明。经过声明之后，在设计中就可以调用库中的代码了。库的声明方式非常简单，如下所示：

```
LIBRAYR library_name;
USE library_name.package_name.package_parts;
```

其中，LIBRARY 和 USE 是 VHDL 保留的关键字。目前设计中常用的是以下 3 个包集：

- ieee.std_logic_1164（来自 ieee 库）
- standard（来自 std 库）
- work（来自 work 库）

它们的声明方式分别为：

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY std;
USE std.standard.all;

LIBRARY work;
USE work.all;
```

其中，分号表示一个语句和声明的结束。当出现“--”时，表示后续文字为注释。

可见，库的声明通常需要使用两条语句，以 ieee 库的声明为例，第一个语句 LIBRARY ieee 声明设计中使用 ieee 库；第二个语句 USE ieee.std_logic_1164.all 声明使用 ieee 库的 std_logic_1164 包集中的所有内容。

注意，std 库和 work 库在程序中都是默认“可见”的，因此不再需要对它们进行声明；只有 ieee 库在使用前需要进行明确的声明。

2.3　实体

实体（ENTITY）用来描述电路的所有输入/输出引脚，其语法结构如下：

```
ENTITY entity_name IS
    PORT (
        port_name: signal_mode signal_type;
        port_name: signal_mode signal_type;
        ...);
END entity_name;
```

端口的信号模式（signal_mode）是以下 4 种之一：IN，OUT，INOUT 或 BUFFER。如图 2.3 所示，IN 和 OUT 是单向引脚，而 INOUT 是双向引脚。BUFFER 模式的引脚首先是一个输出引脚，但该输出信号可以供本电路内部使用。需要注意的是，OUT 类型的端口是不能供电路内部使用的。信号的类型包括 BIT，STD_LOGIC 和 INTEGER 等，第 3 章将对此进行详细讨论。最后要说明一点，ENTITY 名称的选取没有严格的规定，但应注意不要与 VHDL 保留的关键字发生冲突。

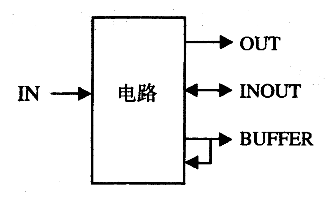


图 2.3 信号模式

例 图 2.4 是一个基本的与非门，我们可以这样描述它的 ENTITY：

```
ENTITY nand_gate IS
   PORT (a, b: IN BIT;
         x: OUT BIT);
END nand_gate;
```

图 2.4 与非门

从上面的 ENTITY 可以看出：电路有 3 个外部引脚，两个输入（a 和 b）和一个输出（x），它们都是 BIT 类型的。ENTITY 的名称是 nand_gate。

2.4 构造体

构造体（ARCHITECTURE）中的代码用来描述电路行为和实现功能，其语法结构如下：

```
ARCHITECTURE architecture_name OF entity_name IS
    [declarations]
BEGIN
    (code)
END architecture_name;
```

从语法结构中可以看到，一个 ARCHITECTURE 包含两个部分：声明部分（可选），用于对信号和常量等进行声明；代码部分（BEGIN 和 END 之间的部分），用来描述电路的行为（功能）。与 ENTITY 一样，可以采用除了 VHDL 关键字以外的任何名称为 ARCHITECTURE 命名，并且允许和 ENTITY 具有相同的名称。

例 图 2.4 所示与非门的 ARCHITECTURE 编写如下：

```
ARCHITECTURE myarch OF nand_gate IS
BEGIN
   x <= a NAND b;
END myarch;
```

从上面的 ARCHITECTURE 中很容易看出：ARCHITECTURE 的名称是 myarch，它所描述的电路功能是对输入的 a 与 b 进行“与非”逻辑运算，运算结果赋予输出信号 x。需要注意的是，在本例中没有出现 ARCHITECTURE 的声明部分，代码部分也只包含了一条语句。

2.5　例题

本节将分析两段简单的 VHDL 代码例题。目前还没有学习 VHDL 的语法知识，但通过分析下面两个例题会对 VHDL 代码结构的基本知识有总体上的认识。在每个例子后面逐行进行了解释说明，并给出了仿真结果。

例 2.1　带有异步复位端的 D 触发器

图 2.5 给出了一种具有异步复位端（rst）并采用时钟（clk）上升沿触发的 D 触发器（DFF）。当 rst = '1'时，无论时钟是什么状态，D 触发器的输出（q）都将被置为低电平。否则，只要时钟信号出现上升沿，输入的值就传递给输出（也就是说，D 触发器是靠时钟的上升沿触发的）。有许多方法可以实现图 2.5 中的 D 触发器，下面给出了其中一种解决方法。有一点必须明确：尽管 VHDL 内部是并发执行的（相对于顺序执行的计算机程序而言），但是要实现时序电路（随着时钟节拍一步步顺序工作的电路），必须使用顺序执行的代码。下例中进程（PROCESS）内部的语句就是顺序执行的。

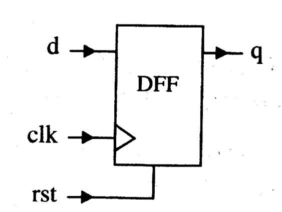


图 2.5　带有异步复位端的 D 触发器

```
1  ---------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  ENTITY dff  IS
6     PORT (d, clk, rst: IN STD_LOGIC;
7           q: OUT  STD_LOGIC);
8  END  dff;
9  ---------------------------------------------------
10 ARCHITECTURE behavior OF dff IS
11 BEGIN
12    PROCESS (rst, clk)
13    BEGIN
14      IF (rst= '1') THEN
15         q <= '0';
```

```
16        ELSIF (clk'EVENT AND clk= '1') THEN
17           q <= d;
18        END IF;
19    END PROCESS;
20 END behavior;
21 --------------------------------------------------
```

注释：

第 2 行~第 3 行：库声明。注意，另外两个必需的库（std 和 work）是默认可见的，不必进行声明。

第 5 行~第 8 行：ENTITY dff。

第 10 行~第 20 行：ARCHITECTURE behavior。

第 6 行：输入端口列表（输入端口的信号模式只能是 IN）。本例中的所有输入信号都属于 STD_LOGIC 类型。

第 7 行：输出端口列表（输出端口的信号模式可以是 OUT，INOUT 或 BUFFER）。本例中的输出信号也是 STD_LOGIC 类型的。

第 11 行~第 19 行：ARCHITECTURE 的代码部分（从 BEGIN 开始）。

第 12 行~第 19 行：PROCESS（PROCESS 内部的代码是顺序执行的）。

第 12 行：每当敏感信号列表中的信号发生变化时，PROCESS 就执行一次。在本例中，每当 rst 和 clk 信号发生变化时，进程就执行一次。

第 14 行~第 15 行：无论 clk 是什么状态，只要 rst = '1'，输出都将复位（异步复位）。

第 16 行~第 17 行：如果 rst 是低电平，且 clk 出现上升沿（clk = '1'），则输入信号将存储在 D 触发器中（q <= d）。

第 15 行和第 17 行："<="操作符用来给信号（SIGNAL）赋值，而":="用来给变量（VARIABLE）赋值。默认情况下，ENTITY 中所有的端口都是信号。

第 1 行、第 4 行、第 9 行和第 21 行：注释行（注意"--"表示注释行）。使用注释可以使代码易于理解。

注意，VHDL 是不区分大小写的。

仿真结果：

图 2.6 给出了例 2.1 的仿真结果。仿真结果清晰易懂，第一列是出现在 ENTITY 中的信号名称，根据箭头可以看出信号的模式。注意 rst 和 clk 的箭头是向内的，内嵌的字母"I"表示其为输入引脚；而 q 的箭头是向外的，内嵌的字母"O"表示其为输出引脚。第二列显示了当前垂直光标所在位置的信号的值。图中的垂直光标在 0 ns 处，信号值分别是 1，0，0 和 0。在本例中，信号值只是简单的 0 和 1，但对于矢量来说，值可以是二进制、十进制或十六进制的形式。第三列就是仿真波形。任意改变输入信号（rst，d，clk）的值，仿真器会给出相应的输出结果（q）。比较图 2.6

的仿真结果和电路期望的结果，可以发现两者是一致的。如前一节所述，本书提供的设计实例都可以采用 Xilinx 公司的开发环境（见附录 B）或 Altera 公司的开发环境（见附录 C）在 CPLD/FPGA（见附录 A）上实现。

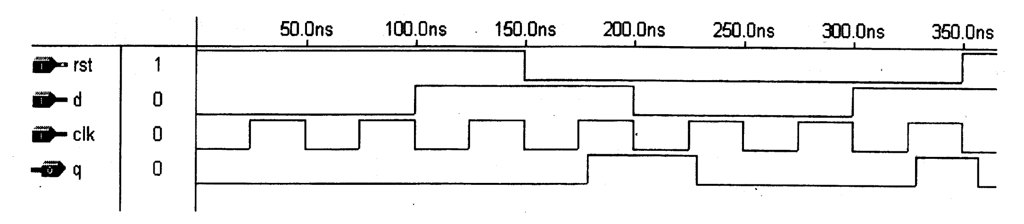


图 2.6　例 2.1 的仿真结果

例 2.2　D 触发器+与非门

图 2.4 是一个纯组合逻辑电路，而图 2.5 是时序逻辑电路。图 2.7 是组合逻辑与时序逻辑相结合的电路（没有复位端）。在下面的分析中，为了说明怎样声明一个信号，我们有意识地引入了一些不必要的（临时）信号。图 2.8 给出了下面代码综合后的仿真结果。

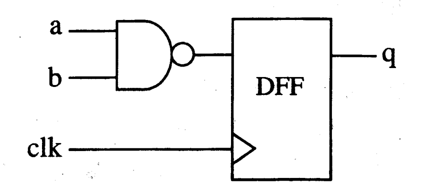


图 2.7　D 触发器和与非门构成的电路

```
1  ---------------------------------------------------
2  ENTITY example IS
3     PORT (a, b, clk: IN BIT;
4           q: OUT BIT);
5  END  example;
6  ---------------------------------------------------
7  ARCHITECTURE example OF example IS
8     SIGNAL  temp: BIT;
9  BEGIN
10    temp <= a NAND b;
11    PROCESS (clk)
12    BEGIN
13       IF (clk'EVENT AND clk= '1') THEN q <= temp;
14       END IF;
15    END PROCESS;
```

```
16 END example;
17 ---------------------------------------------------
```

注释：

这里的端口信号是 BIT 类型的，BIT 数据类型在 std 库中定义。本例中不需要进行库声明（因为 std 库和 work 库是默认可见的）。

第 2 行~第 5 行：ENTITY example。

第 7 行~第 16 行：ARCHITECTURE example。

第 3 行：输入端口（BIT 类型）。

第 4 行：输出端口（BIT 类型）。

第 8 行：ARCHITECTURE 的声明部分（可选）。信号 temp 声明为 BIT 类型。注意，这里没有信号模式说明（只有在 ENTITY 中才出现）。

第 9 行~第 15 行：ARCHITECTURE 的代码部分（从 BEGIN 开始）。

第 11 行~第 15 行：一个 PROCESS（每当信号 clk 发生改变时顺序执行这些语句）。

第 10 行和第 11 行~第 15 行：尽管 PROCESS 内部的语句是顺序执行的，但是 PROCESS 作为一个整体和 PROCESS 外部的其他语句是并发执行的，因此第 10 行和第 11 行~第 15 行是并发执行的。

第 10 行：与非运算，运算结果赋给 temp 信号。

第 13 行~第 14 行：IF 语句，在时钟的上升沿把 temp 的值赋给 q。

第 10行和第 13 行："<="操作符用来给信号(SIGNAL)赋值。而" := "用来给变量(VARIABLE)赋值。

第 8 行和第 10 行：如果将第 13 行改成"q <= a NAND b"，那么这两行就可以省略。

第 1 行、第 6 行和第 17 行：注释行（--符号表示后面的内容为注释），使用注释可以使代码易于理解。

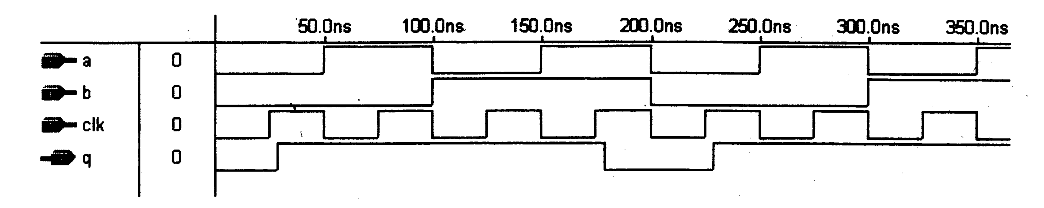


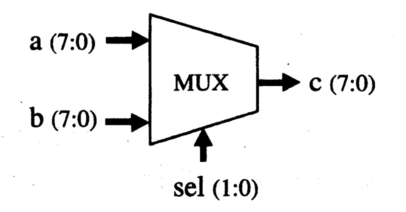
图 2.8　例 2.2 的仿真结果

2.6　习题

2.1　多路选择器

多路选择器的顶层电路如图 P2.1 所示。根据真值表，如果输入 sel = "01"或者 sel = "10"，那

么输出将等于对应的某一个输入（c = a 或 c = b）。然而如果输入 sel = "00"或者 sel = "11"，那么输出将分别为'0'和'Z'（高阻）。



sel	c
00	0
01	a
10	b
11	Z

图 P2.1

（a）填写空格，完成下面的代码。
（b）试对你的解答给出相关的注释（参照例 2.1 和例 2.2）。
（c）将代码编译后进行仿真，验证其正确性。

注意，因为 IF 语句很直观，所以下面的代码使用了它。在以后的学习中将会发现，多路选择器也可以用其他语句（如 WHEN 语句或 CASE 语句）来实现。

```
1  --------------------------------------------------
2  LIBRARY ieee;
3  USE ________________;
4  --------------------------------------------------
5  ENTITY  mux  IS
6      PORT ( ___, ___: ___ STD_LOGIC_VECTOR(7 DOWNTO 0);
7           sel: IN ________________;
8            ___: OUT  STD_LOGIC_ VECTOR(7 DOWNTO 0));
9  END  ______;
10 --------------------------------------------------
11 ARCHITECTURE  example  OF  ___ IS
12 BEGIN
13    PROCESS (a, b, __)
14    BEGIN
15       IF (sel="00") THEN
16          c <= "00000000";
17       ELSIF ( _______ ) THEN
18          c <= a;
19       _____(sel= "10") THEN
20          c <= _;
21       ELSE
22          c <= (OTHERS =>  '__' );
```

```
23          END ___;
24    END ______;
25 END ______;
26 --------------------------------------------------
```

2.2 逻辑门

（a）试编写一段 VHDL 代码，实现图 P2.2 所示的电路。注意，由于要求得到纯组合逻辑电路，所以不需要使用 PROCESS 语句。要求使用逻辑操作符（AND，OR，NAND 和 NOT 等）写出 d 的表达式。

（b）综合后进行仿真，在确定它正确工作后，查看报告文件中由编译器生成的 d 的实际表达式并和所编写的表达式进行比较。

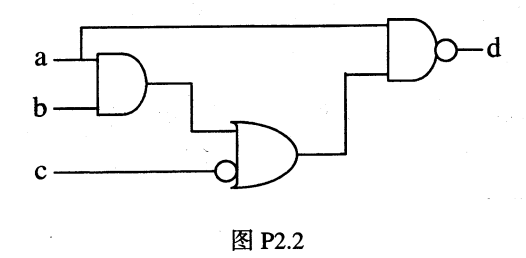


图 P2.2

第3章　数据类型

为了更有效地编写 VHDL 代码，必须知道哪些数据类型是可用的以及怎样说明和使用它们。本章将学习 VHDL 的所有基本数据类型，尤其是那些可以综合的数据类型。此外，本章对数据类型的兼容性和数据类型的转换也进行了介绍。

3.1　预定义的数据类型

在 IEEE 1076 和 IEEE 1164 标准中包括一系列预定义的数据类型。这些数据类型可以在下面的包集/库中找到更详细的描述。

- std 库的 standard 包集：定义了位（BIT）、布尔（BOOLEAN）、整数（INTEGER）和实数（REAL）数据类型。
- ieee 库的 std_logic_1164 包集：定义了 STD_LOGIC 和 STD_ULOGIC 数据类型。
- ieee 库的 std_logic_arith 包集：定义了 SIGNED 和 UNSIGNED 数据类型。还定义了 conv_integer(p)，conv_unsigned(p, b)，conv_signed(p, b)和 conv_std_logic_vector(p, b)等数据类型转换函数。
- ieee 库的 std_logic_signed 和 std_logic_unsigned 包集：包含一些函数，这些函数可以使 STD_LOGIC_VECTOR 类型的数据进行像 SIGNED 和 UNSIGNED 类型数据一样的运算。

下面将对所有预定义的数据类型进行说明。

- 位（BIT）和位矢量（BIT_VECTOR）：位值用'0'或'1'表示。

例

```
SIGNAL x: BIT;
-- 将 x 声明为一个位宽为 1 的 BIT 类型的信号。
SIGNAL y: BIT_VECTOR(3 DOWNTO 0);
--将 y 声明为一个位宽为 4 的位矢量，其中最左边的一位是最高位（MSB：Most Significant Bit）。
SIGNAL w: BIT_VECTOR(0 DOWNTO 7);
-- 将 w 声明为一个位宽为 8 的位矢量，它的最右边的一位是 MSB。
```

在定义了上述信号以后，可以采用下面的方式对信号赋值（必须使用“ <= ”操作符给信号赋值）。

```
x <= '1';
```

```
-- x 是位宽为 1，值为'1'的信号。注意，当位宽为 1 时，位值放在单引号中。
y <= "0111";
-- y 是位宽为 4，值为"0111"（MSB = '0'）的信号。注意，当位宽大于 1 时，位矢量值放在双引号中。
w <= "01110001";
--w 是位宽为 8，值为"01110001"（MSB = '1'）的信号。
```

- STD_LOGIC 和 STD_LOGIC_VECTOR：它们是 IEEE 1164 标准中引入的 8 逻辑值系统。不同于 BIT 数据类型，它可以取'0'，'1'，不定态和高阻态等 8 种不同的值。

```
'X'   "强"不确定值（综合后为不确定值）
'0'   "强"0  （综合后为 0）
'1'   "强"1  （综合后为 1）
'Z'   高阻态（综合后为三态缓冲器）
'W'   "弱"不确定值
'L'   "弱"0
'H'   "弱"1
'-'   不可能出现的情况
```

例

```
SIGNAL x: STD_LOGIC;
-- 声明 x 是位宽为 1 的 STD_LOGIC 类型的信号。
SIGNAL y: STD_LOGIC _VECTOR（3 DOWNTO 0） := "0001";
-- 声明 y 是一个位宽为 4 的矢量，其中最左边的一位是 MSB。
-- 对信号 y 赋初始值"0001"（可选）。注意，使用":="对信号赋初始值。
```

上述 STD_LOGIC 类型的 8 种可能取值中，只有'0'，'1'和'Z'是可综合的，其他 5 种主要用于仿真。在我们的一般观念中，数字逻辑值非'0'即'1'，为什么这里定义了多达 8 种的逻辑值呢？考虑这样一个问题：当两个或两个以上数字逻辑电路的输出端连接在同一个节点上时，这个节点上的电平应该是什么呢？当分析此时节点的电平时，既与两者当前的输出电平值有关，也与两者的驱动能力强弱有关，驱动能力强的电路可以将节点电平强行拉高或拉低。建立多逻辑值系统就是为了对这类情况进行细分。表 3.1 给出了当多个输出连接在同一个节点上时，最终节点电平的取值规定。

表 3.1　8 逻辑值系统数值关系表

	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X

（续表）

	X	0	1	Z	W	L	H	–
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
–	X	X	X	X	X	X	X	X

- STD_ULOGIC 和 STD_ULOGIC_VECTOR：是 IEEE 1164 标准中定义的具有 9 种逻辑值的数据类型（'X'，'0'，'1'，'Z'，'W'，'L'，'H'，'–'和'U'）。事实上，上述的 STD_LOGIC 类型是 STD_ULOGIC 类型的一个子集。后者引入了一个新的逻辑值'U'，它代表初始不定值。与 STD_LOGIC 不同的是，STD_ULOGIC 中没有指定两个 STD_ULOGIC 信号连接到同一个节点上发生冲突后的逻辑值，因此要避免两个输出信号进行直接连接。尽管如此，在确保两根输出线不会连接到一起的条件下，这种 9 逻辑值系统可以用来检测设计时可能发生的错误。
- 布尔类型（BOOLEAN）：只有两种取值，“真”或“假”。
- 整数（INTEGER）：32 位的整数（取值范围从–2 147 483 647 到 2 147 483 647）。
- 自然数（NATURAL）：非负的整数（从 0 到 2 147 483 647）。
- 实数（REAL）：实数的取值范围从-1.0×10^{38}到 1.0×10^{38}，它是不可综合的。
- 物理量字符（Physical literal）：用来表示诸如时间和电压等物理量。在仿真时可以使用，但不可综合。
- 字符（CHARACTER）型：可以是单个或者一串 ASCII 字符。
- SIGNED（有符号数）和 UNSIGNED（无符号数）：它们是在 ieee 库 std_logic_arith 包集中定义的数据类型。从外在表现上看，它们与 STD_LOGIC_VECTOR 相同，但能够支持与整型变量类似的算术运算。我们将在 3.7 节中详细讨论有符号数（SIGNED）类型和无符号数（UNSIGNED）类型。

例

```
x0 <= '0';                -- 可以是 BIT，STD_LOGIC 或者 STD_ULOGIC 类型的值'0'
x1 <= "00011111";         -- 可以是 BIT_VECTOR，STD_LOGIC_VECTOR，
                          -- STD_ULOGIC_VECTOR，SIGNED 或 UNSIGNED 类型的值
x2 <= "0001_1111";        -- 数字之间加下划线可以增加二进制数的可读性
x3 <= "101111";           -- 二进制数"101111"，表示十进制数 47
x4 <= B"101111";          -- 二进制数"101111"，表示十进制数 47
x5 <= O"57";              -- 八进制数"57"，表示十进制数 47
x6 <= X"2F";              -- 十六进制数"2F"，表示十进制数 47
n <= 1200;                -- 整数
m <= 1_200;               -- 整数表示中允许出现下划线
```

```
IF ready THEN ...          -- 布尔运算，如果 ready 为真，就执行 THEN 之后的语句
y <= 1.2E-5;               -- 实数，不可综合
q <= d after 10 ns;        -- 物理量，不可综合
```

例 不同类型数据之间的合法与非法操作：

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR（7 DOWNTO 0）;
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR（7 DOWNTO 0）;
SIGNAL e: INTEGER RANGE 0 TO 255;
...
a <= b(5);        -- 合法（标量类型：BIT）
b(0) <= a;        -- 合法（标量类型：BIT）
c <= d(5);        -- 合法（标量类型：STD_LOGIC）
d(0) <= c;        -- 合法（标量类型：STD_LOGIC）
a <= c;           -- 非法（BIT 和 STD_LOGIC 数据类型不匹配）
b <= d;           -- 非法（BIT_VECTOR 和 STD_LOGIC_VECTOR 数据类型不匹配）
e <= b;           -- 非法（INTEGER 和 BIT_VECTOR 数据类型不匹配）
e <= d;           -- 非法（INTEGER 和 STD_LOGIC_VECTOR 数据类型不匹配）
```

3.2 用户定义的数据类型

VHDL 允许用户自己定义数据类型。下面给出了两种用户定义的数据类型：integer 和 enumerated。

- 用户定义的整数（integer）类型：

```
TYPE integer IS RANGE -2147483647 TO +2147483647;
-- 用户定义的整数类型，与预定义的整数类型是相同的。
TYPE natural IS RANGE 0 TO +2147483647;
-- 用户定义的自然数类型，与预定义的自然数类型是相同的。
TYPE my_integer IS RANGE -32 TO 32;
-- 用户定义的整数类型的子集。
TYPE student_grade IS RANGE 0 TO 100;
-- 用户定义的自然数类型的子集。
```

- 用户定义的枚举（enumerated）类型：

```
TYPE bit IS ('0', '1');
-- 与预定义的 BIT 数据类型在本质上是相同的。
TYPE my_logic IS ('0', '1', 'Z');
```

```
-- 用户定义的 STD_LOGIC 类型的子类。
TYPE bit_vector IS ARRAY(NATURAL RANGE< >) OF BIT;
-- 用户定义的 BIT_VECTOR 数据类型。
-- RANGE< >表示数据取值范围没有约束。
-- NATURAL RANGE< >表示数据值约束在自然数范围内。
TYPE state IS (idle, forward, backward, stop);
-- 枚举数据类型，常用于有限状态机的定义。
TYPE color IS (red, green, blue, white);
-- 另一种枚举类型的定义。
```

一般来说，枚举类型的数据自动按顺序依次编码（除非在用户自定义属性中进行了特别说明，关于用户自定义属性的问题将在第 4 章中进行讨论）。例如，上述自定义的枚举类型 color 会采用两个位按顺序编码的方式："00"表示第一个状态（red），"01"表示第二个状态（green），"10"表示第三个状态（blue），"11"表示第四个状态（white）。

3.3 子类型

在原有已定义数据类型基础上加一些约束条件，可以定义该数据类型的子类型。通过定义新的数据类型可以达到同样的目的，但是要注意，VHDL 不允许不同类型的数据之间直接进行操作运算，而某个数据类型的子类型则可以和原有类型数据直接进行操作运算。

例　给前面例子中出现的数据类型加上一些约束条件后，得到下面的子类型：

```
SUBTYPE natural IS INTEGER RANGE 0 TO INTEGER'HIGH;
-- 定义自然数（NATURAL）为整数（INTEGER）的子类型。
SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO 'Z';
-- 我们知道：STD_LOGIC=('X', '0', '1', 'Z', 'W', 'H', 'L', '-'),
-- 因此有 my_logic=('0', '1', 'Z')。
SUBTYPE my_color IS color RANGE red TO blue;
-- 因为 color =(red, green, blue, white)，所以 my_color=(red, green, blue)。
SUBTYPE  small_integer IS INTEGER RANGE -32 TO 32;
-- 整数类型的子类型。
```

例　子类型和原有类型数据之间的合法和非法操作如下所示：

```
SUBTYPE my_logic IS STD_LOGIC RANGE '0'TO'1';
SIGNAL a: BIT;
SIGNAL b: STD_LOGIC;
SIGNAL c: my_logic;
...
b <= a;  -- 非法（BIT 和 STD_LOGIC 数据类型不匹配）。
```

```
b <= c;  -- 合法（c 是在 STD_LOGIC 数据类型加上约束条件后的子类型，b 是 STD_LOGIC 数据
            类型）。
```

3.4 数组

数组（Array）是将相同数据类型的数据集合在一起形成的一种新的数据类型。它可以是一维（1D）的，也可是二维（2D）的或者 1×1 维（1D×1D）的。更高维数的数组通常是不可综合的。

数组的结构如图 3.1 所示。图中（a）是一个标量，（b）是一个矢量，也是一个一维数组，（c）是一个矢量数组，也是一个 1×1 维的数组，（d）是一个二维标量数组。

事实上，可以认为 VHDL 预定义的数据类型（见 3.1 节）仅包括标量类型（单个位）和矢量类型（一维数组）两类。这两类中只有下列数据类型是可以综合的：

- 标量：BIT，STD_LOGIC，STD_ULOGIC 和 BOOLEAN。
- 矢量：BIT_VECTOR，STD_LOGIC_VECTOR，STD_ULOGIC_VECTOR，INTEGER，SIGNED 和 UNSIGNED。

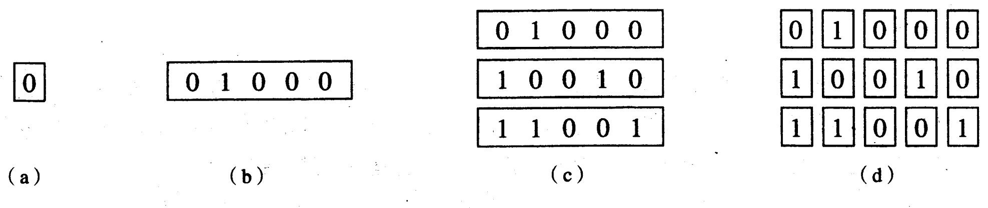


图 3.1　（a）标量；（b）1D；（c）1D×1D；（d）2D

我们注意到，没有预定义二维和 1×1 维的数组，用户可以根据需要自己进行定义。为此，我们先要用 TYPE 命令定义一种新的数据类型，进而可以用这种新的数据类型对信号、变量或常量进行声明。定义新的数组类型采用如下语法结构：

```
TYPE type_name IS ARRAY (specification ) OF data_type;
```

使用新的数组类型对信号、变量和常数进行声明，可以采用如下语法结构：

```
SIGNAL signal_name: type_name [ := initial_value];
```

在上面的语法结构中，定义了一个新的数组类型的信号。采用同样的方法，可以定义数组类型的常量（CONSTANT）和变量（VARIABLE）。注意，上面的初始值是可选的。

例　1×1 维的数组

要求建立一个数组，它包含 4 个矢量，每个矢量位宽为 8。这实际上是一个 1×1 维的数组，参见图 3.1（c）。我们称每个矢量为“行”，称整个数组为“矩阵”。此外，称每个矢量最左边的位为 MSB，最上面的一行为“第 0 行”。那么数组可以这样来实现（注意例子中的信号 x 是阵列类型的）：

```
TYPE row IS ARRAY(7 DOWNTO 0) OF STD_LOGIC;          -- 一维数组
TYPE matrix IS ARRAY(0 TO 3) OF row;                 -- 1×1 维数组
SIGNAL x: matrix;                                    -- 1×1 维信号
```

例 另一种 1×1 维的数组

可以用另一种方法来构造上例中的 1×1 维数组：

```
TYPE matrix IS ARRAY(0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

从数据兼容性的角度来看，后一种方法比前一种更具有优势（参见例 3.1）。

例 二维数组

下面给出了一个二维数组，它并不是由矢量构成的，而是由标量构成的。

```
TYPE matrix2D IS ARRAY(0 TO 3, 7 DOWNTO 0) OF STD_LOGIC; -- 二维数组
```

例 数组的初始化

在上面给出的语法结构中，对信号和变量赋初始值的操作是可选的。我们可以按如下方式对信号和变量赋初始值。

```
... := "0001";                                              -- 对一维数组初始化
... := ('0', '0', '0', '1');                                -- 对一维数组初始化
... := (('0', '1', '1', '1'), ('1', '1', '1', '1') );       -- 对 1×1 维数组或二维
                                                               数组初始化
```

例 合法和非法的数组赋值

```
TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;       -- 一维数组
TYPE array1 IS ARRAY (0 TO 3) OF row;              -- 1×1 维数组
TYPE array2 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
                                                   -- 1×1 维数组
TYPE array3 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
                                                   -- 二维数组
SIGNAL x: row;
SIGNAL y: array1;
SIGNAL v: array2;
SIGNAL w: array3;
```

下面的数组赋值操作是基于上面定义的数据类型和信号声明的。

```
--------------------合法的标量赋值---------------------
-- 因为所有信号（x，y，v 和 w）都源于 STD_LOGIC 类型，所以下面的标量（单个位）赋值都是合
   法的。
x(0) <= y(1)(2);                    -- 注意有两个圆括号（y 是 1×1 维数组）
```

```
x(1) <= v(2)(3);                 -- 注意有两个圆括号（v是1×1维数组）
x(2) <= w(2, 1);                 -- 注意只有一个圆括号（w是二维数组）
y(1)(1) <= x(6);
y(2)(0) <= v(0)(0);
y(0)(0) <= w(3, 3);
w(1, 1) <= x(7);
w(3, 0) <= v(0)(3);
--------------------------矢量赋值------------------
x <= y(0);                       -- 合法（同是ROW类型的）
x <= v(1);                       -- 非法，类型不匹配（x和v(1)分别是ROW和
                                 -- STD_LOGIC_VECTOR类型的）
x <= w(2);                       -- 非法（w必须带有两个索引值）
x <= w(2, 2 DOWNTO 0);           -- 非法（x是ROW类型的，而右侧是STD_LOGIC类型的）
v(0) <= w(2, 2 DOWNTO 0);        -- 非法（V(0)是STD_LOGIC_VECTOR类型的，而右侧是
                                 -- STD_LOGIC类型的，数据类型不匹配）
v(0) <= w(2);                    -- 非法（w必须带有两个索引值）
y(1) <= v(3);                    -- 非法（y(1)和v(3)分别是ROW和STD_LOGIC_VECTOR类
                                 -- 型的）
y(1)(7 DOWNTO 3) <= x(4 DOWNTO 0);         -- 合法（同类型，同宽度）
v(1)(7 DOWNTO 3) <= v(2)(4 DOWNTO 0);      -- 合法（同类型，同宽度）
w(1, 5 DOWNTO 1) <= v(2)(4 DOWNTO 0);      -- 非法（数据类型不匹配）
```

3.5 端口数组

根据前面所学的内容，可以发现预定义的数据类型都没有超过一维。然而，在定义电路的输入/输出端口时，有时需要把端口定义为矢量阵列。由于在 ENTITY 中不允许使用 TYPE 进行类型定义，所以必须在包集中根据端口的具体信号特征建立用户自定义的数据类型，该数据类型可以供包括 ENTITY 在内的整个设计使用。下面给出一个例子：

```
-----------------------包集------------------------
LIBRARY  ieee;
USE  ieee.std_logic_1164.all;
-----------------------------
PACKAGE my_data_types IS
   TYPE vector_array IS ARRAY (NATURAL RANGE <>) OF
     STD_LOGIC_VECTOR(7 DOWNTO 0);
END my_data_types;
-----------------------主代码----------------------
LIBRARY  ieee;
```

```
USE  ieee.std_logic_1164.all;
USE  work.my_data_types.all;   -- 用户定义的包集
--------------------------------------------------
ENTITY mux IS
    PORT (inp: IN vector_array(0 TO 3);
     ...);
END mux;
     ...;
--------------------------------------------------
```

在上面的例子中定义了一个名为 vector_array 的二维数组。它由多个 8 位矢量组成，NATURAL RANGE< >表示数据范围可以自由选择，但是必须在自然数范围内。新的数据类型保存在一个名为 my_data_types 的包集中。在 ENTITY 中，利用这个包集定义了一个名为 inp 的端口。注意，在代码开始部分加入了一个 USE 子句，声明了 my_data_type 包集，这样用户就可以在下面的代码中使用包集 my_data_types。

对于上面例子中的包集，也可以采用如下方式建立，这里包含一个常量的声明。第 10 章将详细讨论包集。

```
------------------ -------包集--------------------
LIBRARY  ieee;
USE  ieee.std_logic_1164.all;
--------------------------------------------------
PACKAGE my_data_types IS
    CONSTANT b: INTEGER := 7;
    TYPE vector_array IS ARRAY (NATURAL RANGE <>) OF
       STD_LOGIC_VECTOR(b DOWNTO 0);
END my_data_types;
--------------------------------------------------
```

3.6　记录类型

记录类型和数组类型有些相似，两者惟一的区别在于 RECORD 类型内部可以包含不同类型的数据，而 ARRAY 只能包含相同类型的数据。

例

```
TYPE birthday IS RECORD
   day: INTEGER RANGE 1 TO 31;
   month: month_name;
END RECORD;
```

3.7 有符号数和无符号数

前面曾提到，在 ieee 库中有一个名为 std_logic_arith 的包集，其中包括有符号（SIGNED）数和无符号数（UNSIGNED）两种数据类型。从下面的例子中可以看到它们的使用方法。

例

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

注意，有符号数和无符号数与整数的语法结构不同，但和 STD_LOGIC_VECTOR 的语法结构相似。

无符号数只能表示大于等于零的数。例如，"0101"表示十进制数 5，"1101"表示十进制数 13。而有符号数则可以表示正数和负数。例如，"0101"表示十进制数 5，"1101"表示十进制数–3。

只有在代码开始部分声明 ieee 库中的包集 std_logic_arith，才能使用有符号数和无符号数。事实上，使用有符号数和无符号数主要是为了进行算术运算（也就是说，与 STD_LOGIC_VECTOR 类型不同，它支持算术运算操作），但是它们不支持逻辑运算。

注意，SIGNED 和 UNSIGNED 类型的数据也支持比较运算操作。

例 有符号数和无符号数的合法与非法操作

```
LIBRARY  ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;  -- 必须声明这个包集才能使用 SIGNED 和 UNSIGNED 数
...
SIGNAL a: IN SIGNED (7 DOWNTO 0);
SIGNAL b: IN SIGNED (7 DOWNTO 0);
SIGNAL x: OUT SIGNED (7 DOWNTO 0);
...
x <= a+b;                          -- 合法（支持算术运算）
x <= a AND b;                      -- 非法（不支持逻辑运算）
```

例 STD_LOGIC_VECTOR 的合法和非法操作

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;       -- 不需要声明其他包集
...
SIGNAL a:IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
```

```
x <= a+b;                              -- 非法（不支持算术运算）
x <= a AND b;                          -- 合法（支持逻辑运算）
```

上面提到 STD_LOGIC_VECTOR 类型的数据不能直接进行算术运算。为了解决这个问题，ieee 库提供了 std_logic_signed 和 std_logic_unsigned 两个包集。声明了这两个包集以后，STD_LOGIC_VECTOR 类型的数据可以像 SIGNED 和 UNSIGNED 类型的数据一样进行算术运算。

例　STD_LOGIC_VECTOR 类型数据的算术运算操作

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;          -- 必须声明这个包集
...
SIGNAL a: IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
...
x <= a+b;                                 -- 合法（支持算术运算）
x <= a AND b;                             -- 合法（支持逻辑运算）
```

3.8　数据类型转换

在 VHDL 中，不同类型的数据不能直接进行算术或逻辑运算。因此有必要进行数据类型转换操作。有两种常见的方法可以实现数据类型转换：一种方法是写一段专门用于数据类型转换的 VHDL 代码；另一种方法是调用包集中预定义的数据类型转换函数。在包集 std_logic_1164 中有一些数据类型转换函数，可以直接使用。

例　不同类型数据的合法与非法操作

```
TYPE long IS INTEGER RANGE -100 TO +100;
TYPE short IS INTEGER RANGE -10 TO +10;
SIGNAL x: short;
SIGNAL y: long;
...
y <= 2*x+5;                  -- 非法（数据类型不匹配）
y <= long(2*x+5);            -- 合法（运算结果已经强制转换成 long 类型）
```

在 ieee 库的包集 std_logic_arith 中提供了许多数据类型转换函数，如下所示：

- conv_integer(p)：将数据类型为 INTEGER，UNSIGNED，SIGNED，STD_ULOGIC 或 STD_LOGIC 的操作数 p 转换成 INTEGER 类型。注意，这里不包含 STD_LOGIC_VECTOR。

- conv_unsigned(p, b)：将数据类型为 INTEGER，UNSIGNED，SIGNED 或 STD_ULOGIC 的操作数转换成位宽为 b 的 UNSIGNED 类型的数据。
- conv_signed(p, b)：将数据类型为 INTEGER，UNSIGNED，SIGNED 或 STD_ULOGIC 的操作数 p 转换成位宽为 b 的 SIGNED 类型的操作数。
- conv_std_logic_vector (p, b)：将数据类型为 INTEGER，UNSIGNED，SIGNED 或 STD_LOGIC 的操作数 p 转换成位宽为 b 的 STD_LOGIC_VECTOR 类型的操作数。

例　数据类型转换

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
...
SIGNAL a: IN UNSIGNED(7 DOWNTO 0);
SIGNAL b: IN UNSIGNED(7 DOWNTO 0);
SIGNAL y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
...
y <= CONV_STD_LOGIC_VECTOR((a+b), 8);
-- 合法，(a+b)被转换成位宽为 8 的 STD_LOGIC_VECTOR 值，然后赋给 y。
```

前一节曾提到，库 ieee 提供了 std_logic_signed 和 std_logic_unsigned 两个包集，在声明了这两个包集以后，STD_LOGIC_VECTOR 类型的数据可以像 SIGNED 和 UNSIGNED 类型的数据一样进行算术运算。

除了上面给出的数据类型转换函数，许多综合工具还给出了其他数据类型转换函数。

3.9 小结

表 3.2 总结了 VHDL 中基本的可综合的数据类型。

表 3.2　可综合的数据类型

数据类型	可综合的数值
BIT，BIT_VECTOR	'0'，'1'
STD_LOGIC，STD_LOGIC_VECTOR	'X'，'0'，'1'，'Z'
STD_ULOGIC，STD_ULOGIC_VECTOR	'X'，'0'，'1'，'Z'
BOOLEAN	True，False
NATURAL	0 到+2 147 483 647
INTEGER	−2 147 483 647 到+2 147 483 647
SIGNED	−2 147 483 647 到+2 147 483 647
UNSIGNED	0 到+2 147 483 647

（续表）

数据类型	可综合的数值
用户自定义整型	INTEGER 的子集
用户自定义枚举类型	根据用户定义进行编码得到
SUBTYPE	任何预定义或用户自定义类型的子集
ARRAY	任意上述单一类型数据的集合
RECORD	任意上述多种类型数据的集合

3.10　例题

本章最后特别给出了一些例子，以说明数据类型的用法。

例 3.1　数据类型用法举例

在给出数据类型定义和信号声明的基础上，下面列出了一些合法和非法的赋值操作：

```
TYPE byte IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;                  -- 一维数组
TYPE mem1 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;          -- 二维数组
TYPE mem2 IS ARRAY (0 TO 3) OF byte;                           -- 1×1 维数组
TYPE mem3 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(0 TO 7);       -- 1×1 维数组
SIGNAL a: STD_LOGIC;                                           -- 标量信号
SIGNAL b: BIT;                                                 -- 标量信号
SIGNAL x: byte;                                                -- 一维信号
SIGNAL y: STD_LOGIC_VECTOR(7 DOWNTO 0);                        -- 一维信号
SIGNAL v: BIT_VECTOR(3 DOWNTO 0);                              -- 一维信号
SIGNAL z: STD_LOGIC_VECTOR(x'HIGH DOWNTO 0);                   -- 一维信号
SIGNAL w1: mem1;                                               -- 二维信号
SIGNAL w2: mem2;                                               -- 1×1 维信号
SIGNAL w3: mem3;                                               -- 1×1 维信号
----------------------合法的标量赋值------------------------------------
x(2) <= a;                              -- 同类型(STD_LOGIC)，使用正确
y(0) <= x(0);                           -- 同类型(STD_LOGIC)，使用正确
z(7) <= x(5);                           -- 同类型(STD_LOGIC)，使用正确
b <= v(3);                              -- 同类型(BIT)，使用正确
w1(0, 0) <= x(3);                       -- 同类型(STD_LOGIC)，使用正确
w1(2, 5) <= y(7);                       -- 同类型(STD_LOGIC)，使用正确
w2(0)(0) <= x(2);                       -- 同类型(STD_LOGIC)，使用正确
w2(2)(5) <= y(7);                       -- 同类型(STD_LOGIC)，使用正确
w1(2, 5) <= w2(3)(7);                   -- 同类型(STD_LOGIC)，使用正确
----------------------非法的标量赋值------------------------------------
b <= a;                          -- BIT 和 STD_LOGIC 数据类型不匹配
```

```
w1(0)(2) <= x(2);              -- w1 的索引必须是二维的
w2(2, 0) <= a;                 -- w2 的索引必须是 1×1 维的
---------------------合法的矢量赋值-------------------------------------
x <= "11111110";
y <= ('1', '1', '1', '1', '1', '1', '0', 'Z');
z <= "11111"&"000";
x <= (OTHERS => '1');
y <= (7 => '0', 1 => '0', OTHERS => '1');
z <= y;
y(2 DOWNTO 0) <= z(6 DOWNTO 4);
w2(0)(7 DOWNTO 0) <= "11110000";
w3(2) <= y;
z <= w3(1);
z(5 DOWNTO 0 ) <= w3(1)( 2 TO 7);
w3(1) <="00000000";
w3(1) <= (OTHERS => '0');
w2 <= ((OTHERS => '0'), (OTHERS => '0'), (OTHERS => '0'), (OTHERS => '0'));
w3 <= ("11111100", ('0', '0', '0', '0', 'z', 'z', 'z', 'z'),(OTHERS => '0'),
       (OTHERS => '0'));
w1 <= ((OTHERS => 'z'), "11110000", "11110000", (OTHERS => '0'));
----------------------合法的数组赋值-------------------------------------
x <= y;                              -- 数据类型匹配
y (5 TO 7 ) <= z(6 DOWNTO 0);        -- y 的索引方向(5 TO 7)错误
w1 <= (OTHERS => '1');               -- w1 是二维数组
w1(0, 7 DOWNTO 0) <= "11111111";     -- w1 是二维数组
w2 <= (OTHERS => 'z');               -- w2 是 1×1 维数组
w2(0, 7 DOWNTO 0) <= "11110000";     -- w2 指数必须是 1×1
----------------------对数组进行初始化的一个实例---------------------------
FOR i IN 0 TO 3 LOOP
   FOR j IN 7 DOWNTO 0 LOOP
        x(j) <= '0';
        y(j) <= '0';
        z(j) <= '0';
        w1(i, j) <= '0';
        w2(i)(j) <= '0';
        w3(i)(j) <= '0';
END LOOP;
END LOOP;
------------------------------------------------------------------------
```

例 3.2　单个位和位矢量

这个例子说明了对单个位赋值和对位矢量赋值的区别（也就是说，对 BIT 和 BIT_VECTOR，STD_LOGIC 和 STD_LOGIC_VECTOR，STD_ULOGIC 和 STD_ULOGIC_VECTOR 的差别进行了对比）。

在下面给出的两段 VHDL 代码中，都对输入信号进行“与”运算，然后将运算结果赋给输出信号。它们之间的惟一区别是输入和输出的位宽不一样（第一段代码中输入和输出的位宽是 1，第二段代码中输入和输出的位宽是 4）。图 3.2 给出了与代码相对应的电路结构。

```
---------------------------
ENTITY and2 IS
     PORT (a, b: IN BIT;
            x: OUT BIT);
END  and2;
---------------------------
ARCHITECTURE and2 OF and2 IS
BEGIN
   x <= a AND b;
END and2;
---------------------------
```

```
-----------------------------------------
ENTITY and2 IS
    PORT (a, b: IN BIT_VECTOR(0 TO 3);
           x: OUT BIT_VECTOR(0 TO 3));
END  and2;
-----------------------------------------
ARCHITECTURE and2 OF and2 IS
BEGIN
   x <= a AND b;
END and2;
-----------------------------------------
```

图 3.2　例 3.2 的代码所对应的电路图

例 3.3　加法器

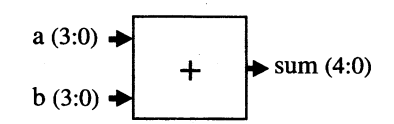


图 3.3　例 3.3 的代码所对应的 4 位加法器

图 3.3 是一个 4 位加法器电路，它有两个输入（a 和 b）和一个输出（sum）。为了实现这个电路，给出了两种解决方法。在前一种方法中，所有信号都是 SIGNED 类型的，而在后一种方法中，所有信号都是 INTEGER 类型的。注意，在后一种方法中，a+b 和 sum 的数据类型不匹配，所

以在第 13 行使用了数据类型转换函数。由于代码中要使用 SIGNED 数据类型，在代码开始部分声明了 std_logic_arith 包集（第 4 行）。应注意，SIGNED 类型的数据在书写上与 STD_LOGIC_VRCTOR 类型的数据相似，而与 INTEGER 类型的数据不同。

```
1  --------------方案 1: in/out=SIGNED---------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  USE  ieee.std_logic_arith.all;
5  ------------------------------------------------
6  ENTITY adder1 IS
7      PORT (a, b: IN SIGNED(3 DOWNTO 0);
8              sum: OUT SIGNED(4 DOWNTO 0));
9  END  adder1;
10 ------------------------------------------------
11 ARCHITECTURE adder1 OF adder1 IS
12 BEGIN
13     sum <= a+b;
14 END adder1;
15 ------------------------------------------------
```

```
1 ----------------方案 2: out=INTEGER----------------
2 LIBRARY  ieee;
3 USE  ieee.std_logic_1164.all;
4 USE  ieee.std_logic_arith.all;
5 ------------------------------------------------
6 ENTITY adder2 IS
7     PORT (a, b: IN SIGNED(3 DOWNTO 0);
8             sum: OUT INTEGER RANGE -16 TO 15);
9 END adder2;
10 ------------------------------------------------
11 ARCHITECTURE adder2  OF adder2 IS
12 BEGIN
13     sum <= CONV_INTEGER(a+b);
14 END adder2;
15 ------------------------------------------------
```

图 3.4 给出了两种解决方案的仿真结果。注意，图中的数值均用两位十六进制数的补码表示。因为输入信号的范围是–8~7，其对应关系是：7→7，6→6，…，0→0，–1→15，–2→14，…，–8→8。同理，输出信号的范围是–16~15，它的对应关系是 15→15，…，0→0，–1→31，…，–16→16。

因此，可以得出：2H + 4H = 0H（即 2 + 4 = 6），4H + 8H = 1CH（即 4 + (−8) = −4）等，这里 H 代表十六进制。

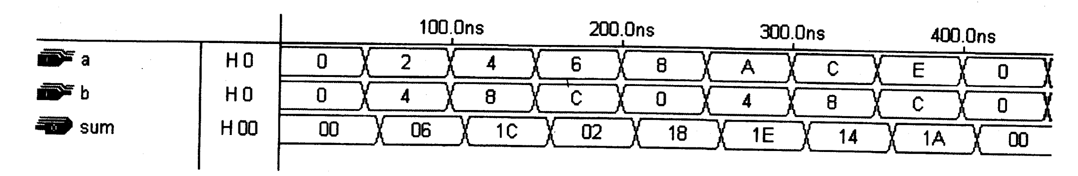


图 3.4　例 3.3 的仿真结果

3.11　习题

以下习题都是基于下面的类型定义和信号说明的：

```
TYPE array1 IS ARRAY (7 DOWNTO 0) OF  STD_LOGIC;
TYPE array2 IS ARRAY (3 DOWNTO 0, 7 DOWNTO 0 ) OF STD_LOGIC;
TYPE array3 IS ARRAY (3 DOWNTO 0) OF array1;
SIGNAL a: BIT;
SIGNAL b: STD_LOGIC;
SIGNAL x: array1;
SIGNAL y: array2;
SIGNAL w: array3;
SIGNAL z: STD_LOGIC_VECTOR(7 DOWNTO 0);
```

3.1　试确定上面给出的信号的维数（标量，一维，二维或 1×1 维），并给出一个属于该数据类型的具体数值。

3.2　试判断表 P3.2 中的赋值操作是否合法并简单解释，同时判断每个操作符两端操作数的维数。

3.3　子类型。为预定义的数据类型 INTEGER 和 STD_LOGIC_VECTOR、上面用户自定义的类型 array1 和 array2 分别建立一个子类型。

3.4　ROM。试用 1×1 维常数来实现只读存储器 ROM（read-only memory），假设一个 ROM 由许多深度为 8，位宽为 4 的块组成。提示：首先建立一个名为 rom 的数组，然后定义一个 rom 类型的信号来实现 ROM，用常数值填充到 ROM 块中：CONSTANT my_rom: rom := (values);。

3.5　简易加法器。重新编写一段代码，实现例 3.3 所示的加法器，要求所有输入/输出信号的类型均为 STD_LOGIC_VECTOR（提示：回顾 3.8 节所学的内容）。

表 P3.2

赋值	赋值符号两侧信号的维数	是否合法及原因
a <= x(2);		
b <= x(2);		
b <= y(3, 5);		
b <= w(5)(3);		
y(1)(0) <= z(7);		
x(0) <= y(0, 0);		
x <= "1110000";		
a <= "0000000";		
y(1) <= x;		
w(0) <= y;		
w(1) <= (7 => '1', OTHERS => '0');		
y(1) <= (0 => '0', OTHERS => '1');		
w(2)(7 DOWNTO 0) <= x;		
w(0)(7 DOWNTO 6) <= z(5 DOWNTO 4);		
x(3) <= x(5 DOWNTO 5);		
b <= x(5 DOWNTO 5);		
y <= ((OTHERS => '0'), (OTHERS => 0), (OTHERS => '0'), "10000001");		
z(6) <= x(5);		
z(6 DOWNTO 4) <= x(5 DOWNTO 3);		
z(6 DOWNTO 4) <= y(5 DOWNTO 3);		
y(6 DOWNTO 4) <= z(3 TO 5);		
y(0, 7 DOWNTO 0) <= z;		
w(2, 2) <= '1';		

第 4 章　运算操作符和属性

本章和前面的几章都是 VHDL 语言的基础。在这个基础上，从下一章开始就可以开展实际的电路设计工作了。VHDL 语法基础的学习令人感到枯燥乏味，但只有在对数据类型、运算操作符及其属性有了深刻理解之后，才有可能写出高质量和高效率的代码。

本章最后给出了一些设计实例。由于本章讲述的仍是 VHDL 语言的基础知识，所以正如前几章，这里给出的都是一些简单的说明性的小例子。正如前面提到的，从第 5 章起才开始进入实质性的电路设计阶段。

4.1　运算操作符

VHDL 提供了 6 种预定义的运算操作符，分别是：

- 赋值运算符
- 逻辑运算符
- 算术运算符
- 关系运算符
- 移位运算符
- 并置运算符

下面将详细介绍每种运算符。

赋值运算符

赋值运算符用来给信号、变量和常数赋值。赋值运算符包括以下 3 种：

<=　　用于对 SIGNAL 赋值。

:=　　用于对 VARIABLE，CONSTANT 和 GENERIC 赋值，也可用于赋初始值。

=>　　给矢量中的某些位赋值，或对某些位之外的其他位（常用 OTHERS 表示）赋值。

例　首先定义下列信号和变量：

```
SIGNAL x: STD_LOGIC;
VARIABLE y: STD_LOGIC_VECTOR(3 DOWNTO 0);       -- 最左边的位是 MSB
SIGNAL w: STD_LOGIC_VECTOR(0 TO 7);             -- 最右边的位是 MSB
```

对于上面的信号或变量，下面的赋值方式都是符合规则的：

```
x <= '1';                                -- 通过 <= 将值'1'赋给信号 x
y :=  "0000";                            -- 通过 := 将值"0000"赋给变量 y
w <= "10000000";                         -- 最低位是 1，其他位是 0
w <= (0 => '1', OTHERS => '0');          -- 最低位是 1，其他位是 0
```

逻辑运算符

逻辑运算符用来执行逻辑运算操作。操作数必须是 BIT，STD_LOGIC 或 STD_ULOGIC 类型的数据（或者是这些数据类型的扩展，即 BIT_VECTOR，STD_LOGIC_VECTOR 或 STD_ULOGIC_VECTOR）。VHDL 的逻辑运算符有以下几种：

- NOT——取反
- AND——与
- OR——或
- NAND——与非
- NOR——或非
- XOR——异或

XNOR 是“同或”运算符，它在 VHDL 87 中没有定义，在 VHDL 93 中被引入。注意，从上至下，这些运算符的优先级是递减的。下面的几个例子使用了逻辑运算符，注意它们的逻辑运算次序：

例

```
y <= NOT a AND b;          -- (a'.b)，这里的“'”表示取反操作
y <= NOT(a AND b);         -- (a.b)'
y <= a NAND b;             -- (a.b)'
```

算术运算符

算术运算符用来执行算术运算操作。操作数可以是 INTEGER，SIGNED，UNSIGNED 或 REAL 数据类型，其中 REAL 类型是不可综合的。如果声明了 ieee 库中的包集 std_logic_signed 和 std_logic_unsigned，即可对 STD_LOGIC_VECTOR 类型的数据进行加法和减法运算（见 3.6 节）。VHDL 语言有以下 8 种算术运算符：

+	加
–	减
*	乘
/	除
**	指数运算
MOD	取模
REM	取余

ABS　　取绝对值

上述运算符中，加法、减法和乘法运算符是可以综合成逻辑电路的，对于除法运算，只有在除数为 2 的 n 次幂时才有可能进行综合，此时除法操作对应的是将被除数向右进行 n 次移位。对于指数运算，只有当底数和指数都是静态数值（常量或 GENERIC 参数）时才是可综合的。在算术运算符的使用中，要注意 MOD 和 REM 的区别：y MOD x 运算的结果是 y 除以 x 所得的余数，运算结果通过信号 x 返回；y REM x 运算的结果是 y 除以 x 所得的余数，结果通过信号 y 返回。ABS 运算返回操作数的绝对值。上述后 3 个运算符（MOD，REM 和 ABS）通常是不可综合的。

关系运算符

关系运算符用来对两个操作数进行比较运算，VHDL 有以下 6 种关系运算符：

=　　等于
/=　　不等于
<　　小于
>　　大于
<=　　小于等于
>=　　大于等于

关系运算符左右两边操作数的数据类型必须相同，这些关系运算符适用于前面所讲的所有数据类型。

移位操作符

移位操作符用来对数据进行移位操作，它们是在 VHDL 93 中引入的。其语法结构为：

<左操作数><移位操作符><右操作数>

其中，左操作数必须是 BIT_VECTOR 类型的，右操作数必须是 INTEGER 类型（前面可以加正负号）的。VHDL 中的移位操作符有以下几种：

- sll　逻辑左移　　-- 数据左移，右端空出来的位置填充'0'
- srl　逻辑右移　　-- 数据右移，左端空出来的位置填充'0'
- sla　算术左移　　-- 数据左移，同时复制最右端的位，在数据左移操作后填充在右端空出的位置上
- sra　算术右移　　-- 数据右移，同时复制最左端的位，在数据右移操作后填充在左端空出的位置上
- rol　循环逻辑左移　　-- 数据左移，同时从左端移出的位依次填充到右端空出的位置上
- ror　循环逻辑右移　　-- 数据右移，同时从右端移出的位依次填充到左端空出的位置上

例 令 x <= "01001"，那么：

```
y <= x sll 2;   -- 逻辑左移两位：y <= "00100"
y <= x sla 2;   -- 算术左移两位：y <= "00111"
y <= x srl 3;   -- 逻辑右移三位：y <= "00001"
y <= x sra 3;   -- 算术右移三位：y <= "00001"
y <= x rol 2;   -- 循环逻辑左移两位：y <= "00101"
y <= x srl -2;  -- 等同于逻辑左移两位
```

并置运算符

并置运算符用于位的拼接，其操作数可以是支持逻辑运算的任何数据类型。并置运算符有以下两种：

- &
- (, , ,)

例 使用并置运算符对信号赋值

```
z <= x & "10000000";              -- 如果 x <= '1'，那么 z <= "11000000"
z <= ('1', '1', '0', '0', '0', '0', '0', '0');    -- z <= "11000000"
```

4.2 属性

VHDL 语言中的属性(ATTRIBUTE)语句可以从指定的客体或对象中获得关心的数据或信息，因此可以使 VHDL 代码更灵活。VHDL 中的预定义属性可以划分为两大类：数值类属性和信号类属性。

本节将详细讨论预定义的属性。用户自定义属性将在 4.3 节中讨论。

数值类属性

数值类属性用来得到数组、块或一般数据的相关信息，例如可用来获取数组的长度和数值范围等。

下面是 VHDL 中预定义的可综合的数值类属性：

- d'LOW 返回数组索引的下限值
- d'HIGH 返回数组索引的上限值
- d'LEFT 返回数组索引的左边界值
- d'RIGHT 返回数组索引的右边界值
- d'LENGTH 返回矢量的长度值
- d'RANGE 返回矢量的位宽范围
- d'REVERSE_RANGE 按相反的次序，返回矢量的位宽范围

例　定义信号：

```
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
```

则有：d'LOW = 0，d'HIGH = 7，d'LEFT = 7，d'RIGHT = 0，d'LENGTH = 8，d'RANGE = (7 DOENTO 0)，d'REVERSE_RANGE = (0 TO 7)。

例　定义信号：

```
SIGNAL x: STD_LOGIC_VECTOR(0 TO 7);
```

则下列 4 个 LOOP 语句都是可综合的，并且是等效的：

```
FOR i IN RANGE(0 TO 7)LOOP...
FOR i IN x'RANGE(0 TO 7)LOOP...
FOR i IN RANGE(x'LOW TO x'HIGH)LOOP...
FOR i IN RANGE(0 TO x'LENGTH-1)LOOP...
```

对于枚举类型的信号，VHDL 中定义了如下属性：

- d'VAL(pos)　返回指定位置（pos）的值
- d'POS(value)　给出数值（value），返回其位置序号
- d'LEFTOF(value)　给出数值（value），返回其左侧的值
- d'VAL(row, column)　返回给定行、列位置对应的值

通常，枚举数据类型的属性是不可综合的。

信号类属性

对于信号 s，有以下预定义的属性：

- s'EVENT　如果 s 的值发生了变化，则返回值为布尔量 TRUE，否则返回 FALSE
- s'STABLE　如果 s 保持稳定，没有发生变化，则返回值为布尔量 TRUE，否则返回 FALSE
- s'ACTIVE　如果当前 s = '1'，则返回 TRUE，否则返回 FALSE
- s'QUIET<time>　如果在指定的时间内 s 没有发生变化，则返回 TRUE，否则返回 FALSE
- s'LAST_EVENT　计算上一次事件发生到现在所经历的时间，并返回这个时间值
- s'LAST_ACTIVE　返回最后一次 s = '1'到现在所经历的时间长度值
- s'LAST_VALUE　返回最后一次变化前 s 的值

大部分信号类属性仅用于仿真，但上面给出的前两个信号类属性（s'EVENT 和 s'STABLE）是可以综合的，其中 s'EVENT 最常用。

例 下面给出的4个赋值语句都是可综合的，并且是等效的。如果信号 clk 出现了上升沿，那么将返回 TRUE，即括号中的返回值为 TRUE。

```
IF (clk'EVENT AND clk = '1')...              -- EVENT 搭配 IF 使用
IF (NOT clk'STABLE AND clk = '1')...         -- STABLE 搭配 IF 使用
WAIT UNTIL (clk'EVENT AND clk = '1')...      -- EVENT 搭配 WAIT 使用
IF RISING_EDGE(clk)...                       -- 调用一个函数
```

4.3 用户自定义属性

上面给出的诸如 HIGH，RANGE，EVENT 等都是在 VHDL 87 中预定义的属性，此外 VHDL 也允许用户自定义属性。使用用户自定义属性首先要进行属性声明，其语法格式为：

```
ATTRIBUTE attribute_name: attribute_type;
```

其中，attribute_type 可以是任何数据类型，包括 BIT，INTEGER，STD_LOGIC 和 STD_LOGIC_VECTOR 等。

此后就可以进行属性描述了，具体语法格式如下：

```
ATTRIBUTE attribute_name OF target_name: class IS value;
```

其中，class 可以是数据类型、信号、变量、函数、实体或构造体等。

例

```
ATTRIBUTE number_of_inputs: INTEGER;                  -- 属性声明
ATTRIBUTE number_of_inputs OF nand3: SIGNAL IS 3;     -- 属性描述
   ...
inputs <= nand3'number_of_inputs;                     -- 属性调用，返回值为 3
```

例 枚举类型编码

在一般情况下，对枚举类型数据采用顺序编码的方式。下面是我们定义的枚举类型 color：

```
TYPE color IS (red, green, blue, white);
```

编码后，red = "00"，green = "01"，blue = "10"，white = "11"。使用用户自定义属性后，可以改变这种默认的编码次序，具体方式如下：

```
ATTRIBUTE enum_encoding OF color: TYPE IS "11 00 10 01";
```

这样，原来默认的编码次序就变成了 red = "11"，green = "00"，blue = "10"和 white = "01"。

除了包集以外，可以在代码中其他任何地方进行用户自定义属性的声明。不同的综合工具对此支持程度不同。当综合通不过时，综合工具会忽略它，或者提出警告。

4.4 操作符扩展

上一节提到用户可以自定义属性，用户同样也可以自定义操作符。在 4.1 节中，预定义的算术运算操作符的操作数必须是特定类型（比如整数类型）的数据，例如预定义操作符“+”的操作数不允许是 BIT 类型的。

用户自定义的操作符可以和 VHDL 中预定义的操作符具有相同的名称。例如，我们用“+”定义一种新的加法运算，其操作数为 BIT_VECTOR 类型，这样就对原来的操作符进行了扩展。

例 要求对一个整数和一个 1 位二进制数进行加法运算。首先，构造一个函数（第 11 章将详细讨论函数的构造和使用）。

```
-------------------------------------------------
FUNCTION "+" (a: INTEGER; b: BIT) RETURN INTEGER IS
BEGIN
   IF (b = '1') THEN RETURN a+1;
   ELSE RETURN a;
   END IF;
END "+";
-------------------------------------------------
```

然后就可以调用这个函数了。

```
-------------------------------------------------
SIGNAL inp1, outp: INTEGER RANGE 0 TO 15;
SIGNAL inp2: BIT;
   (...)
outp <= 3+inp1+inp2;
   (...)
-------------------------------------------------
```

在表达式 outp <= 3+inp1+inp2 中，第一个“+”是预定义的加法运算操作符（两个整数相加）。第二个“+”是经过扩展的用户自定义加法运算操作符，它能对一个整数和一个 BIT 类型数据进行加法运算。

4.5 通用属性语句

GENERIC（属性）语句提供了一种指定常规参数的方法，所指定的参数是静态的，设计人员可以根据具体设计需求方便地进行参数修改。该语句可以增加代码的灵活性和可重用性。GENERIC 语句必须在 ENTITY 中进行声明。GENERIC 语句指定的参数是全局的，不仅可以在 ENTITY 内部使用，也可以在后面的整个设计中使用。其语法结构如下：

```
GENERIC (parameter_name: parameter_type := parameter_value);
```

例 下面的 GENERIC 语句指定了一个整数类型的参数 n，其默认值是 8，所以在实体和构造体中只要出现了 n，就认为它的值为 8。

```
ENTITY my_entity IS
   GENERIC (n: INTEGER := 8);
   PORT (...);
END my_entity;
ARCHITECTURE my_architecture OF my_entity IS
   ...
END my_architecture;
```

当然，也可以使用一个 GENERIC 语句指定多个参数，例如：

```
GENERIC (n: INTEGER := 8; vector: BIT_VECTOR := "00001111");
```

下面将给出一些完整的设计实例。这些例子进一步说明了 GENERIC 语句、属性和运算操作符的用法。

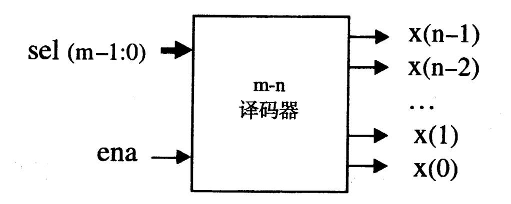
4.6 设计实例

为了进一步说明 GENERIC 语句、属性和运算操作符的用法，在此给出了一些完整的设计实例。尽管在前面学习的基础上，读者已经建立了 VHDL 语言的一些基本概念，但是对于代码编写技术还知之甚少（该内容将在第 5 章开始讨论）。对于 VHDL 语言的初学者来说，在下面的例子中还会遇到一些生疏的语法和结构，这没有关系，读者可以尝试结合前面学习的知识和数字电路的相关知识分析这些例题，以巩固本章所学的内容。在学习完第 5 章和第 7 章之后再回头来看这些实例，将会有更准确的理解。

例 4.1 通用译码器

图 4.1 是一个通用的 m-n（输入为 m 位，输出为 n 位）译码器。电路有两个输入端口和一个输出端口：sel，ena 和 x。假设 n 是 2 的 m 次幂，即 $m=\log_2 n$。如果 ena = '0'，那么 x 的所有输出位都变成高电平，否则 x 的值由图 4.1 给出的真值表确定。

下面给出的构造体完全是通用的，改变第 7 行中 sel 和第 8 行中 x 的位宽，就可以满足不同的设计要求。在本例中取 m = 3，n = 8。可以看出，如果使用 GENERIC 语句来指定 m 和 n 的值，代码会变得更灵活和易于重用。后面的一些例子会经常使用 GENERIC 语句（参见习题 4.4）。



ena	sel	x
0	00	1111
1	00	1110
	01	1101
	10	1011
	11	0111

图 4.1　例 4.1 中的译码器

注意下面代码中操作符的用法："+"（第 22 行）、"*"（第 22 行和第 24 行）、" := "（第 17 行、第 18 行、第 22 行、第 24 行和第 27 行）、" <= "（第 29 行）和 " => "（第 17 行）。注意属性的用法：HIGH（第 14 行和第 15 行）和 RANGE（第 20 行）。

```
 1 ---------------------------------------------
 2 LIBRARY ieee;
 3 USE ieee.std_logic_1164.all;
 4 ---------------------------------------------
 5 ENTITY decoder IS
 6   PORT (ena: IN STD_LOGIC;
 7          sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
 8          x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
 9 END decoder;
10 ---------------------------------------------
11 ARCHITECTURE generic_decoder OF decoder IS
12 BEGIN
13    PROCESS (ena, sel)
14       VARIABLE temp1: STD_LOGIC_VECTOR(x'HIGH DOWNTO 0);
15       VARIABLE temp2: INTEGER RANGE 0 TO x'HIGH;
16    BEGIN
17       temp1 := (OTHERS => '1');
18       temp2 := 0;
19       IF (ena= '1') THEN
20          FOR i IN sel'RANGE LOOP          -- sel 的位宽是 3 (2 DOWNTO 0)
21             IF (sel(i) = '1') THEN        -- 二进制到十进制的转换
22                temp2 := 2*temp2+1;
23             ELSE
24                temp2 := 2*temp2;
25             END IF;
26         END LOOP;
27         temp1(temp2) := '0';
28    END IF;
```

```
29      x <= temp1;
30    END PROCESS;
31  END generic_decoder;
32  ---------------------------------------------------
```

图 4.2 的仿真结果验证了解码器的功能。可以看到，当 ena = '0'时，x 的所有位都是高电平，x = "11111111"（二进制数 255）。当 ena = '1'时，x 中只有一位（由 sel 选择）是低电平的。例如，当 sel = "000"时，x = "11111110"；当 sel = "001"时，x = "11111101"；当 sel = "010"时，x = "11111011"。

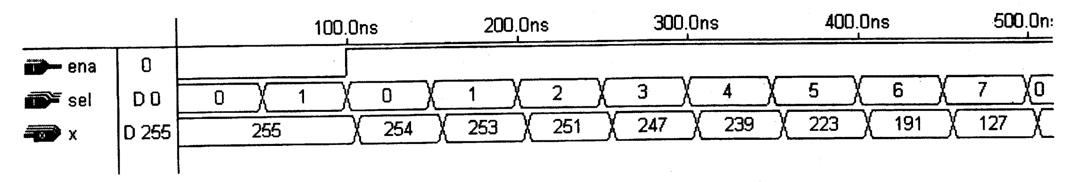


图 4.2 例 4.1 的仿真结果

例 4.2 通用奇偶校验检测器电路

图 4.3 给出了一个奇偶校验检测器的顶层电路图。当输入矢量中'1'的个数是偶数时，电路输出'0'。注意，在下面 VHDL 代码的 ENTITY 中使用了一个 GENERIC 语句（第 3 行），指定 n = 7。只要改变第 3 行中 n 的值 7，这段代码就可以满足不同长度矢量的要求。最后，要求读者把设计中出现的操作符和属性标注出来。

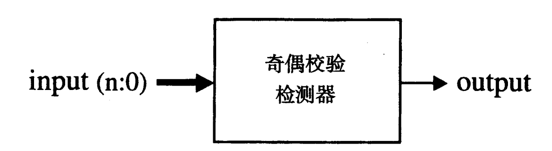


图 4.3 例 4.2 中的通用奇偶校验检测器

```
1  ----------------------------------------------
2  ENTITY parity_det IS
3      GENERIC (n: INTEGER :=  7);
4      PORT ( input: IN  BIT_VECTOR(n DOWNTO 0);
5                output: OUT BIT);
6  END  parity_det;
7  ----------------------------------------------
8  ARCHITECTURE parity OF parity_det IS
9  BEGIN
10     PROCESS (input)
11         VARIABLE  temp: BIT;
12     BEGIN
```

```
13        temp := '0';
14        FOR i IN input'RANGE LOOP
15           temp := temp XOR input(i);
16        END LOOP;
17        output <= temp;
18     END PROCESS;
19 END parity;
20 ----------------------------------------------
```

图 4.4 给出了上述代码的仿真结果。注意，当输入为"00000000"时，由于其中'1'的个数是偶数，所以输出为'0'；当输入为"00000001"时，由于其中'1'的个数是奇数，所以输出为'1'。

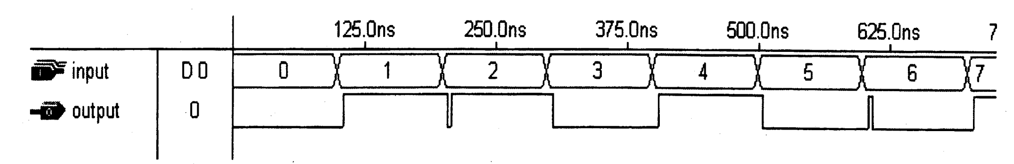


图 4.4　例 4.2 的仿真结果

例 4.3　通用奇偶校验发生器电路

从图 4.5 所示的电路可以看出，其左侧输入矢量的宽度比右侧输出矢量的宽度少 1 位。当输入矢量中'1'的个数分别为奇数和偶数时，所增加的输出位的值相应地为'1'和'0'，这样使得输出矢量中'1'的个数恒为偶数。

下面给出了描述奇偶校验发生器的 VHDL 代码。要求读者把代码中的操作符和属性标注出来。

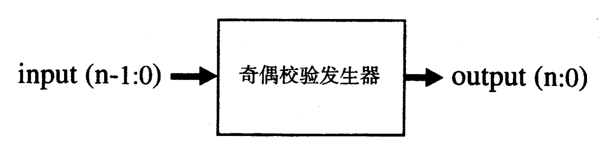


图 4.5　例 4.3 中的奇偶校验发生器电路

```
1 ----------------------------------------------
2 ENTITY parity_gen IS
3    GENERIC (n: INTEGER := 7);
4    PORT ( input: IN  BIT_VECTOR(n-1 DOWNTO 0);
5           output: OUT BIT_VECTOR(n DOWNTO 0));
6 END parity_gen;
7 ----------------------------------------------
8 ARCHITECTURE parity OF parity_gen IS
9 BEGIN
10   PROCESS (input)
```

```
11      VARIABLE  temp1: BIT;
12      VARIABLE  temp2: BIT_VECTOR(output'RANGE);
13    BEGIN
14      temp1 :=  '0';
15      FOR i IN input'RANGE LOOP
16         temp1 := temp1 XOR input(i);
17         temp2(i) := input(i);
18      END LOOP;
19         temp2(output'HIGH ) :=  temp1;
20         output <= temp2;
21    END PROCESS;
22 END parity;
23 --------------------------------------------------
```

图 4.6 给出了上面代码的仿真结果。我们可以看到：当输入为"0000000"（7 位）时，输出为"00000000"（8 位）；当输入为"0000001"（7 位）时，输出为"10000001"（8 位）。

图 4.6　例 4.3 的仿真结果

4.7　小结

表 4.1 和表 4.2 分别对 VHDL 中的操作符和属性进行了总结。注意，标有“◆”表示不可综合（或者在很少的综合工具上是可综合的）。

表 4.1　运算操作符

操作符类型	操作符	操作数类型
赋值运算	<= ， := ， =>	任意数据类型
逻辑运算	NOT，AND，NAND， OR，NOR，XOR，XNOR	BIT，BIT_VECTOR， STD_LOGIC，STD_LOGIC_VECTOR， STD_ULOGIC，STD_ULOGIC_VECTOR
算术运算	+，−，*，/，** (mod，rem，abs)◆	INTEGER，SIGNED，UNSIGNED
比较运算	=，/=，<，>， <= ， >=	任意数据类型
移位运算	sll，srl，sla，sra，rol，ror	BIT_VECTOR
并置运算	&，(，，，)	STD_LOGIC，STD_LOGIC_VECTOR， STD_ULOGIC，STD_ULOGIC_VECTOR， SIGNED，UNSIGNED

表 4.2　属　性

应用场合	属性	返回值
常规类型数据	d'LOW	返回数组索引的下限值
	d'HIGH	返回数组索引的上限值
	d'LEFT	返回数组索引的左边界值
	d'RIGHT	返回数组索引的右边界值
	d'LENGTH	返回矢量的长度值
	d'RANGE	返回矢量的位宽范围
	d'REVERSE_RANGE	按相反的次序，返回矢量的位宽范围
枚举数据类型	d'VAL（pos）◆	返回指定位置（pos）的值
	d'POS(value) ◆	给定数值（value），返回其位置序号
	d'LEFTOF(value) ◆	给定数值（value），返回其左侧的值
	d'VAL(row, column) ◆	返回给定行、列位置对应的值
信号	s'EVENT	如果 s 的值发生了变化，那么返回值为布尔量 TRUE，否则返回 FALSE
	s'STABLE	如果 s 保持稳定，没有发生变化，则返回值为布尔量 TRUE，否则返回 FALSE
	s'ACTIVE◆	如果当前 s = 1，则返回 TRUE，否则返回 FALSE

4.8 习题

首先定义以下信号：

```
SIGNAL a: BIT := '1';
SIGNAL b: BIT_VECTOR(3 DOWNTO 0) := "1100";
SIGNAL c: BIT_VECTOR(3 DOWNTO 0) := "0010";
SIGNAL d: BIT_VECTOR(7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
SIGNAL f: INTEGER RANGE -128 TO 127;
```

4.1 写出信号进行下列各种运算后的结果。

```
x1 <= a&b;                   ->      x1 <= ______
x2 <= c&b;                   ->      x2 <= ______
x3 <= b XOR c;               ->      x3 <= ______
x4 <= a NOR b(3);            ->      x4 <= ______
x5 <= b sll 2;               ->      x5 <= ______
x6 <= b sla 2;               ->      x6 <= ______
x7 <= b rol 2;               ->      x7 <= ______
x8 <= a AND NOT b(0) AND NOT c(1);         --x8 <= ______
d <= (5 =>  '0', OTHERS =>  '1');          --d <= ______
```

4.2 写出下列信号的属性。

```
c'LOW               ->________
d'HIGH              ->________
c'LEFT              ->________
d'RIGHT             ->________
c'RANGE             ->________
d'LENGTH            ->________
c'REVERSE_RANGE     ->________
```

4.3 指出下列运算操作符的使用是否正确，并简要说明理由。

```
b(0) AND a
a+d(7)
NOT b XNOR c
c+d
e-f
IF (b<c) ...
IF (b >= a) ...
IF (f /=e) ...
IF (e>d) ...
b sra 1
c srl -2
f ror 3
e*3
5**5
f/4
e/3
d <= c
d(6 DOWNTO 3) := b
e <= d
f := 100
```

4.4 **通用译码器**。下面这个习题和例 4.1 中的译码器电路有关。

（1）在例 4.1 给出的电路中，如果矢量的位宽发生变化，那么程序中的信号 sel（第 7 行）和 x（第 8 行）的位宽也要相应地改变。现在想要把例 4.1 中的设计修改为一个通用译码器。为此必须在 ENTITY 中使用 GENERIC 语句指定 sel 的位宽（假设 n = 3），然后用 n 的函数来替代 sel 和 x 的位宽上界。综合后，对电路进行仿真，验证其正确性。

（2）在例 4.1 的设计中引入了一个二进制数到整数的转换函数（第 20 行~第 26 行）。如果把 sel 声明为整数，就不需要使用这个转换函数。要求读者修改代码，将信号 sel 声明为整数类型。当信号 sel 的位宽用 n 来指定时，代码才是通用的。综合代码并进行仿真。

4.5 试列出例 4.2 和例 4.3 中用到的所有运算符、属性和 GENERIC 参数。

第 5 章　并 发 代 码

前面四章学习了 VHDL 语言的基本知识，从本章开始将逐步进入 VHDL 代码设计学习阶段。

VHDL 代码按照执行顺序可以分为并发代码和顺序代码两大类。在读者以前学习的 PASCAL 或 C 语言中，语句都是逐行顺序执行的，所以称之为程序。而对于硬件电路来说，情况更复杂一些，所有的逻辑门在任何时刻都处于执行状态，这是与传统计算机语言的根本不同之处。这里的并发代码，有些书上称之为并行代码，是本章的学习重点。在下一章中将讨论顺序代码。

VHDL 中的并发描述语句有 WHEN 和 GENERATE。除此之外，仅包含 AND，NOT，+，*和 sll 等操作符的赋值语句也是并发执行的。另外，在块（BLOCK）中的代码也是并发执行的。

5.1　并发执行和顺序执行

为了更好地学习本章的内容，需要先回顾一下组合逻辑与时序逻辑的区别，然后将它们与并发代码和顺序代码的区别进行比较。

组合逻辑和时序逻辑

根据数字逻辑电路的基本知识，组合逻辑电路当前输出的值仅取决于当前的输入，如图 5.1（a）所示。很显然，这样的电路中不需要触发器等具有存储器能力的逻辑电路单元，仅仅使用组合逻辑门就可以实现整个设计。

对于时序逻辑电路来说，电路的当前输出不仅取决于当前的输入，还与以前的输入有关，如图 5.1（b）所示。时序电路中通常存在寄存器这类元件，当前的输出结果是由当前输入和电路内部所有寄存器的状态共同决定的。如图 5.1（b）所示，这类电路中包括寄存器等元件，也包括组合逻辑电路，寄存器通过一个反馈环和组合逻辑模块相连。可见，这些寄存器的当前状态将会对电路的当前输出产生影响。

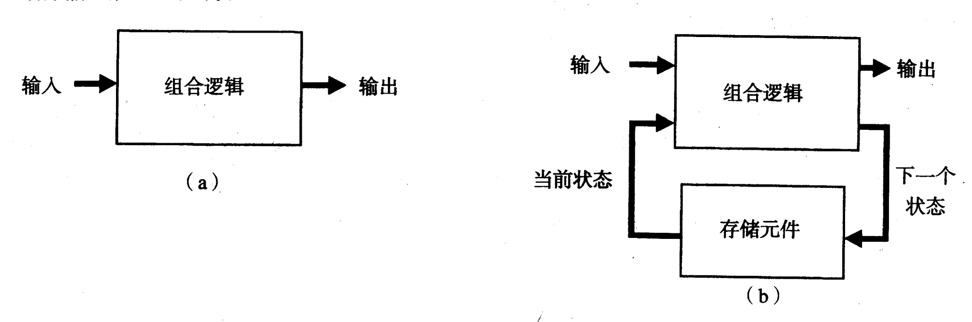


图 5.1　（a）组合逻辑电路；（b）时序逻辑电路

有一点需要注意，不是所有内部包括存储元件的电路都是时序电路。以随机存储器（Random Access Memory，简称 RAM）为例，图 5.2 给出了其电路模型，图中的存储元件与组合逻辑部分不构成反馈环。RAM 的输出值仅仅和当前输入的地址有关。尽管 RAM 带有存储元件，但它不是时序逻辑电路。

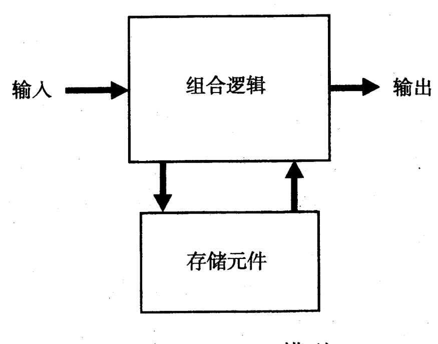


图 5.2　RAM 模型

并发和顺序代码

从本质上讲，VHDL 代码是并发执行的。只有 PROCESS，FUNCTION 或 PROCEDURE 内部的代码才是顺序执行的。值得注意的是，尽管这些模块中的代码是顺序执行的，当它们作为一个整体时，与其他模块之间又是并发执行的。并发代码又称为"数据流"代码。

以一段含有 3 个并发描述语句（stat1，stat2 和 stat3）的代码为例。无论对下面 3 个语句怎样排序，都会产生同样的电路结构。

```
stat1      stat3      stat1
stat2  ≡   stat2  ≡   stat3  ≡其他排列顺序
stat3      stat1      stat2
```

既然上述语句综合产生的电路与代码的书写顺序无关，那么很显然，仅仅使用并发描述语句是无法实现同步时序电路的（GUARDED BLOCK 是惟一的例外，有关 GUARDED BLOCK 的内容将在后面的章节中学习）。换言之，通常我们只能用并发描述语句来实现组合逻辑电路。为了实现时序逻辑电路，必须使用顺序描述语句（见第 6 章）。事实上，使用顺序描述语句可以同时实现组合逻辑电路和时序逻辑电路。

本章讨论的是并发代码，也就是说要学习的是那些在 PROCESS，FUNCTION 和 PROCEDURE 之外使用的语句，即 WHEN 和 GENERATE 语句。除此之外，只使用逻辑、算术等运算操作符的赋值语句也可以产生组合逻辑电路。另外，在块（BLOCK）中的代码也是并发执行的。

综上所述，在并发代码中可以使用下列各项：

- 运算操作符
- WHEN 语句（WHEN/ELSE 或者 WITH/SELECT/WHEN）

- GENERATE 语句
- BLOCK 语句

5.2　使用运算操作符

使用运算操作符是建立并发代码的最基本方法。表 5.1 总结了 4.1 节中讨论的 AND，OR，+，–，*，sll 和 sra 等运算操作符。

运算操作符可以用来实现任何组合逻辑电路。需要注意的是，通过后面的章节可以了解到，只有将时序逻辑与组合逻辑进行有机的结合才能更有效地实现复杂的电路。下面给出的是一个仅使用逻辑运算符的设计实例。

表 5.1　运算操作符

运算类型	运算操作符	操作数类型
逻辑运算	NOT，AND，NAND， OR，NOR，XOR，XNOR	BIT，BIT_VECTOR， STD_LOGIC，STD_LOGIC_VECTOR， STD_ULOGIC，STD_ULOGIC_VECTOR
算术运算	+，–，*，/，** (mod，rem，abs)	INTEGER，SIGNED，UNSIGNED
比较运算	=，/=，<，>，<=，>=	任意数据类型
移位运算	sll，srl，sla，sra，rol，ror	BIT_VECTOR
并置运算	&，(，，，)	STD_LOGIC，STD_LOGIC_VECTOR， STD_ULOGIC，STD_ULOGIC_VECTOR， SIGNED，UNSIGNED

例 5.1　多路复用器#1

图 5.3 给出了一个四输入的多路复用器，它根据选择位 s1 和 s0 的值，选择其中的一路输出。下面，我们仅使用逻辑运算操作符来实现多路复用器。

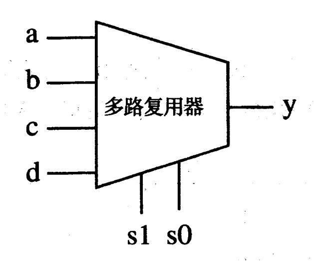


图 5.3　例 5.1 中的多路复用器

```
1 ---------------------------------------------------
2 LIBRARY  ieee;
3 USE  ieee.std_logic_1164.all;
4 ---------------------------------------------------
5 ENTITY mux IS
6     PORT ( a, b, c, d, s0, s1: IN  STD_LOGIC;
7             y: OUT STD_LOGIC);
8 END  mux;
9 ---------------------------------------------------
10 ARCHITECTURE pure_logic OF mux IS
11 BEGIN
12     y <= (a AND NOT s1 AND NOT s0) OR
13          (b AND NOT s1 AND s0) OR
14          (c AND s1 AND NOT s0) OR
15          (d AND s1 AND s0);
16 END pure_logic;
17 ---------------------------------------------------
```

图 5.4 给出了电路功能的仿真结果。

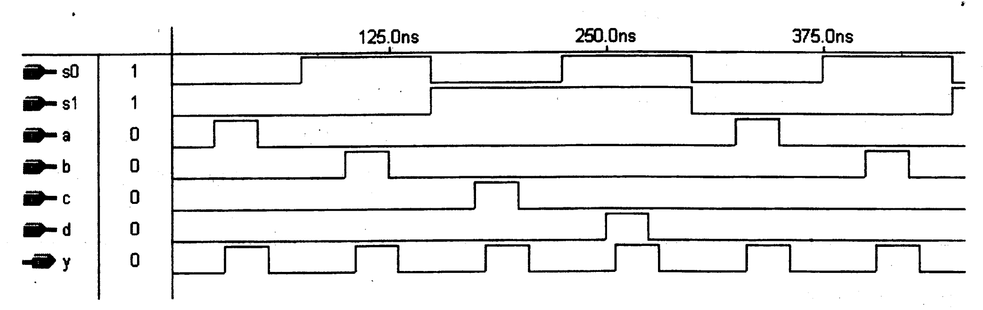


图 5.4　例 5.1 的仿真结果

5.3　WHEN 语句

如前所述，WHEN 语句是一种基本的并发描述语句。它有两种形式：WHEN/ELSE（又称为 simple WHEN）和 WITH/SELECT/WHEN（又称为 selected WHEN）。其语法结构如下：

WHEN/ELSE：

```
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...;
```

WITH/SELECT/WHEN：

```
WITH identifier SELECT
assignment WHEN value,
assignment WHEN value,
...;
```

当使用 WITH/SELECT/WHEN 时，必须对所有可能出现的条件（condition）给予考虑，所以经常使用关键字 OTHERS。如果在某些条件出现时不需要进行任何操作，那么应该使用 UNAFFECTED 指出这一点。下面的例子可以帮助我们更好地理解 WHEN 语句的使用。

例

```
----------------with WHEN/ELSE-------------------
outp <= "000" WHEN (inp = '0' OR reset = '1') ELSE
       "001" WHEN ctl = '1' ELSE
       "010";

--------------with WITH/SELECT/WHEN--------------
WITH control SELECT
   output <= "000" WHEN reset,
        "111" WHEN set,
        UNAFFECTED  WHEN OTHERS;

-------------------------------------------------
```

在上面的例子中，当使用 WITH/WHEN/ELSE 时，语句 UNAFFECTED WHEN OTHERS 是非常重要的，说明在 control 取其他值时 output 的值保持不变。

关于 WHEN 语句还有一点很重要，那就是 WHEN value 的描述方式包括以下 3 种：

```
WHEN value                  -- 针对单个值进行判断
WHEN value1 to value2       -- 针对取值范围进行判断，适用于枚举类型
WHEN value1 | value2 | …    -- 针对多个值进行判断
```

例 5.2　多路复用器#2

这个例子实现了和例 5.1 中一样的多路复用器，但是输入信号 sel 和例 5.1 稍有不同（见图 5.5）。在本例中使用了 WHEN 语句而不是单纯的逻辑运算符。下面给出了两种实现方案，一种使用 WHEN/ELSE 语句（simple WHEN），另一种使用 WITH/SELECT/WHEN 语句（selected WHEN）。这两种实现方式得到的实验结果和例 5.1 是相似的。

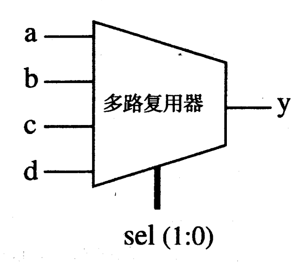


图 5.5　例 5.2 中的多路复用器

```
1  -------------方案 1: with WHEN/ELSE---------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN  STD_LOGIC;
7              sel: IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
8              y: OUT STD_LOGIC);
9  END  mux;
10 -------------------------------------------------
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13     y <= a WHEN sel="00" ELSE
14           b WHEN sel="01" ELSE
15           c WHEN sel="10" ELSE
16           d;
17 END mux1;
18 -------------------------------------------------
1  ----------方案 2: with WITH/SELECT/WHEN------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN  STD_LOGIC;
7              sel: IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
8              y: OUT STD_LOGIC);
9  END  mux;
10 -------------------------------------------------
11 ARCHITECTURE mux2 OF mux IS
12 BEGIN
```

```
13   WITH sel SELECT
14      y <= a WHEN "00",   -- 注意使用","而不是";"
15           b WHEN "01",
16           c WHEN "10",
17           d WHEN OTHERS; -- 不能是 d WHEN "11";
18 END mux2;
19 --------------------------------------------------
```

在上面的实现方案中，sel 信号也可以声明为 INTEGER 类型，此时实现多路复用器的代码如下：

```
1  --------------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN  STD_LOGIC;
7            sel: IN  INTEGER RANGE 0 TO 3;
8            y: OUT STD_LOGIC);
9  END  mux;
10 -------------方案1: with WHEN/ELSE---------------
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13    y <= a WHEN sel = 0 ELSE
14         b WHEN sel = 1 ELSE
15         c WHEN sel = 2 ELSE
16         d;
17 END mux1;
18 ----------方案2: with WITH/SELECT/WHEN----------
19 ARCHITECTURE mux2 OF mux IS
20 BEGIN
21    WITH sel SELECT
22       y <= a WHEN 0,
23            b WHEN 1,
24            c WHEN 2,
25            d WHEN 3;   -- 这里用'3'和 OTHERS 是等效的
26 END mux2;          -- sel 的所有情况都应该考虑
27 --------------------------------------------------
```

注意，上面的两种实现方式分别出现在不同的构造体中，由于在电路实现时一次只能对一个构造体进行综合，所以在电路综合前应该将当前不需要综合的构造体以注释的方式进行屏蔽，或者修

改综合过程中使用的脚本文件，以达到屏蔽的目的。在仿真过程中，可以使用 CONFIGURATION（配置）语句来选择所需的构造体。

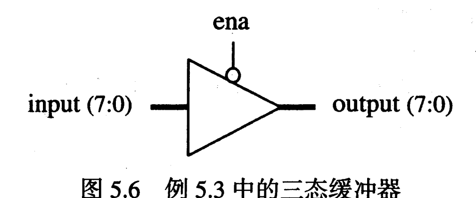


图 5.6 例 5.3 中的三态缓冲器

例 5.3 三态缓冲器

这是用来说明 WHEN 用法的另一个例子。图 5.6 给出了三态缓冲器的具体电路。当信号 ena 是低电平时，输出等于输入，反之输出为"ZZZZZZZZ"（高阻态）。

```
1  LIBRARY  ieee;
2  USE  ieee.std_logic_1164.all;
3  ---------------------------------------------
4  ENTITY tri_state IS
5     PORT (ena: IN  STD_LOGIC;
6              input: IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
7              output: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
8  END  tri_state;
9  ---------------------------------------------
10 ARCHITECTURE tri_state OF tri_state IS
11 BEGIN
12    output <= input WHEN (ena = '0')  ELSE
13              (OTHERS => 'Z');
14 END tri_state;
15 ---------------------------------------------
```

上述代码综合后的仿真结果如图 5.7 所示。和预期的结果一样，当信号 ena 是低电平时，输出等于输入；当信号 ena 是高电平时，输出为"ZZZZZZZZ"（高阻态）。

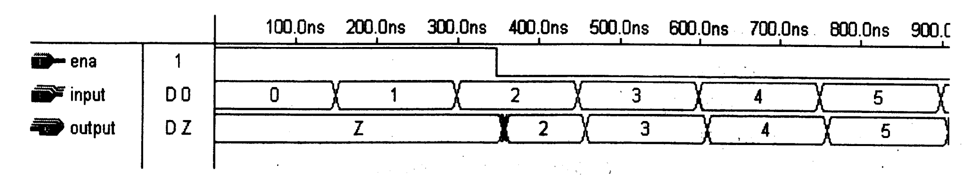


图 5.7 例 5.3 的仿真结果

例 5.4 编码器

图 5.8 给出了一个 n×m 编码器的顶层电路图。我们假设 n 与 m 符合 $m = \log_2 n$ 的关系。每次

只有一个输入是高电平的，它将被编码后输出。下面给出了两种实现方案：一种使用 WHEN/ELSE 语句（simple WHEN），另一种使用 WITH/SELECT/WHEN 语句（select WHEN）。

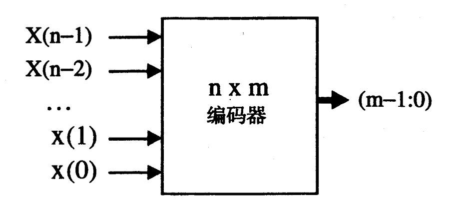


图 5.8 例 5.4 中的编码器

```
1  ------------方案 1: with WHEN/ELSE--------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY encoder IS
6     PORT ( x: IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
7            y: OUT  STD_LOGIC_VECTOR(2 DOWNTO 0));
8  END  encoder;
9  ---------------------------------------------
10 ARCHITECTURE encoder1 OF encoder IS
11 BEGIN
12    y <= "000" WHEN x="00000001" ELSE
13         "001" WHEN x="00000010" ELSE
14         "010" WHEN x="00000100" ELSE
15         "011" WHEN x="00001000" ELSE
16         "100" WHEN x="00010000" ELSE
17         "101" WHEN x="00100000" ELSE
18         "110" WHEN x="01000000" ELSE
19         "111" WHEN x="10000000" ELSE
20         "ZZZ" ;
21 END encoder1;
22 ---------------------------------------------
```

```
1  ------------方案 2: with WITH/SELECT/WHEN----------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  ---------------------------------------------
```

```
5  ENTITY encoder IS
6     PORT ( x: IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
7            y: OUT  STD_LOGIC_VECTOR(2 DOWNTO 0));
8  END  encoder;
9  -----------------------------------------------------
10 ARCHITECTURE encoder2 OF encoder IS
11 BEGIN
12    WITH x SELECT
13       y <= "000" WHEN "00000001",
14            "001" WHEN "00000010",
15            "010" WHEN "00000100",
16            "011" WHEN "00001000",
17            "100" WHEN "00010000",
18            "101" WHEN "00100000",
19            "110" WHEN "01000000",
20            "111" WHEN "10000000",
21            "ZZZ" WHEN OTHERS;
22 END encoder2;
23 -----------------------------------------------------
```

注意，上面的代码中有一长串测试列表（方案 1 中的第 12 行~第 20 行，方案 2 中的第 13 行~第 21 行）。当选择信号的位数增加时，这个列表会变得很长，在这种情况下可以使用 GENERATE 语句（见 5.4 节）和 LOOP 语句（见 6.6 节）进行描述。

上面代码的仿真结果如图 5.9 所示。

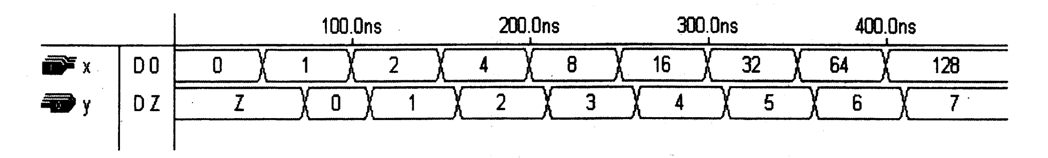
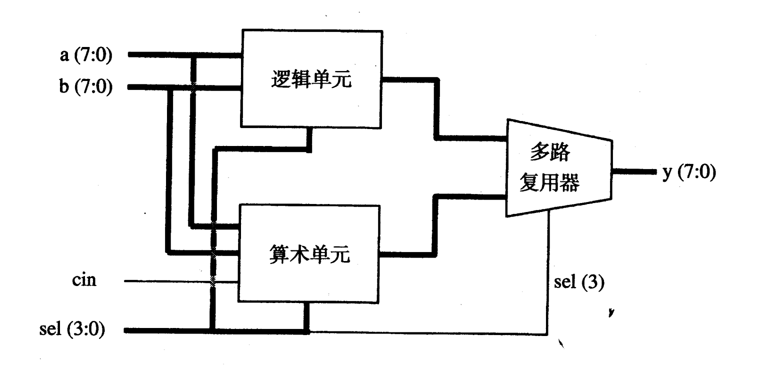


图 5.9　例 5.4 的仿真结果

例 5.5　ALU（算术逻辑单元）

图 5.10 给出的是一个算术逻辑单元。正如其名称一样，它是一种可以执行算术（Arithmetic）运算操作和逻辑（Logic）运算操作的电路。图 5.10 给出了电路的真值表。信号 sel 的最高位用来选择输出哪一种运算（算术运算或逻辑运算）结果，信号 sel 的另外 3 位用来选择执行哪一种运算。



sel	操作	功能	电路单元
0000	y<=a	选择 a	算术单元
0001	y<=a+1	将 a 加 1	
0010	y<=a−1	将 a 减 1	
0011	y<=b	选择 b	
0100	y<=b+1	将 b 加 1	
0101	y<=b−1	将 b 减 1	
0110	y<=a+b	将 a 与 b 相加	
0111	y<=a+b+cin	将 a，b 与进位相加	
1000	y<=NOTa	将 a 取反	逻辑单元
1001	y<=NOTb	将 b 取反	
1010	y<=a AND b	a 和 b 进行逻辑“与”运算	
1011	y<=a OR b	a 和 b 进行逻辑“或”运算	
1100	y<=a NAND b	a 和 b 进行逻辑“与非”运算	
1101	y<=a NOR b	a 和 b 进行逻辑“或非”运算	
1110	y<=a XOR b	a 和 b 进行逻辑“异或”运算	
1111	y<=a XNOR b	a 和 b 进行逻辑“同或”运算	

图 5.10　例 5.5 中的 ALU

下面的代码涉及并发描述语句的用法和同种数据类型条件下算术运算和逻辑运算的实现方法。为了进行算术运算，代码最开始就声明了库 ieee 中的包集 std_logic_unsigned（参见 3.6 节）。在代码中，使用 arith 和 logic 这两个信号来分别表示算术运算和逻辑运算的结果。经过多路复用器的选择后，输出其中一个运算结果。图 5.11 给出了具体的仿真结果。

图 5.11　例 5.5 的仿真结果

```
1 ---------------------------------------------
2 LIBRARY  ieee;
3 USE  ieee.std_logic_1164.all;
4 USE  ieee.std_logic_unsigned.all;
5 ---------------------------------------------
6 ENTITY ALU IS
7    PORT ( a, b: IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
8                 sel: IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
9                 cin: IN  STD_LOGIC;
10                y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
11 END  ALU;
12 ---------------------------------------------
13 ARCHITECTURE dataflow OF ALU IS
14    SIGNAL arith, logic: STD_LOGIC_VECTOR(7 DOWNTO 0);
15 BEGIN
16    ---------------Arirhmetic unit: -------------
17    WITH sel (2 DOWNTO 0) SELECT
18       arith <= a  WHEN "000",
19                a+1 WHEN "001",
20                a-1 WHEN "010",
21                b   WHEN "011",
22                b+1 WHEN "100",
23                b-1 WHEN "101",
24                a+b WHEN "110",
25                a+b+cin WHEN OTHERS;
26  ------------------Logic unit: -----------------
27  WITH sel (2 DOWNTO 0) SELECT
28     logic <= NOT a     WHEN "000",
29              NOT b    WHEN "001",
30              a AND b  WHEN "010",
31              a OR b    WHEN "011",
32              a NAND b WHEN "100",
33              a NOR b  WHEN "101",
34              a XOR b  WHEN "110",
35              NOT(a XOR b)  WHEN OTHERS;
36  --------------------Mux: ---------------------
37  WITH sel (3) SELECT
38     y <= arith WHEN '0',
39          logic WHEN OTHERS;
```

```
40 END dataflow;
41 --------------------------------------------------
```

5.4 GENERATE 语句

GENERATE 语句是另一种基本的并发描述语句。它和顺序描述语句中的 LOOP 语句一样用于循环执行某项操作，通常与 FOR 一起使用。具体的语法结构如下：

FOR/GENERATE：

```
label: FOR identifier IN range GENERATE
  (concurrent assignments)
END GENERATE;
```

GENERATE 语句还有另一种形式：IF/GENERATE，这里不允许使用 ELSE。IF/GENERATE 可以嵌套在 FOR/GENERATE 内部使用。同样，FOR/GENERATE 也可以嵌套在 IF/GENERATE 中使用。

IF/GENERATE 嵌套在 FOR/GENERATE 中：

```
label1: FOR identifier IN range GENERATE
   ...
   label2: IF condition GENERATE
      (concurrent assignments)
   END GENERATE;
   ...
END GENERATE;
```

例

```
SIGNAL x: BIT_VECTOR(7 DOWNTO 0);
SIGNAL y: BIT_VECTOR(15 DOWNTO 0);
SIGNAL z: BIT_VECTOR(7 DOWNTO 0);
...
G1: FOR i IN x'RANGE GENERATE
   z(i) <= x(i) AND y(i+8);
END GENERATE;
```

注意，GENERATE 中循环操作的上界和下界都必须是静态的。如果上界或下界中的参数是非静态的，那么代码通常是不可综合的。在下面的代码中，choice 是一个输入参数，因为上界 choice 是非静态的，所以这样的代码通常是不可综合的。

```
NotOK: FOR i IN 0 TO choice GENERATE
   (并发描述语句)
```

```
END GENERATE;
```

在 GENERATE 语句使用过程中，容易出现多值驱动问题，下面代码是可以正常综合的：

```
OK: FOR i IN 0 TO 7 GENERATE
   output(i) <= '1' WHEN (a(i) AND b(i))= '1' ELSE '0';
END GENERATE;
```

然而，当出现下面两种情况之一时，编译器会提示多驱动错误，同时停止编译。

```
NotOK: FOR i IN 0 TO 7 GENERATE
    accum <= "11111111" WHEN (a(i) AND b(i))= '1'  ELSE "00000000";
END GENERATE;
NotOK: FOR i IN 0  TO 7 GENERATE
    accum <= accum +1 WHEN x(i)  = '1';
END GENERATE;
```

例 5.6 矢量移位器

这个例子可以很好地说明 GENERATE 语句的用法。在本例中，输出矢量是输入矢量进行移位操作后的结果，并且输出矢量位宽是输入的两倍，移位操作的次数由一个输入信号指定。例如，如果输入总线的位宽为 4，值为"1111"，那么输出必定是下面阵列中的某一行（输入矢量用下划线标注）。

row(0): 0 0 0 0 1 1 1 1
row(1): 0 0 0 1 1 1 1 0
row(2): 0 0 1 1 1 1 0 0
row(3): 0 1 1 1 1 0 0 0
row(4): 1 1 1 1 0 0 0 0

第一行和输入相比较没有移位，而且高 4 位用 0 填充。后面每一行都是前一行左移 1 位的结果。

在下面的代码中，inp 为输入矢量，outp 为输出矢量，sel 为移位选择信号。第 14 行中数组（matrix）的每一行都是一个 vector 子类型的矢量（参见第 12 行）。

```
1  ----------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  ----------------------------------------------
5  ENTITY shifter IS
6     PORT ( inp: IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
7            sel: IN  INTEGER RANGE 0 TO 4;
8            outp: OUT  STD_LOGIC_VECTOR(7 DOWNTO 0));
```

```
9  END  shifter;
10 ---------------------------------------------------
11 ARCHITECTURE shifter OF shifter IS
12     SUBTYPE vector IS  STD_LOGIC_VECTOR(7 DOWNTO 0);
13     TYPE matrix  IS ARRAY (4 DOWNTO 0) OF VECTOR;
14     SIGNAL row: matrix;
15 BEGIN
16     row(0) <= "0000"&inp;
17     G1: FOR i IN 1 TO 4 GENERATE
18        row(i) <= row(i-1)(6 DOWNTO 0) & '0';
19     END GENERATE;
20     outp <= row(sel);
21 END shifter;
22 ---------------------------------------------------
```

图 5.12 给出了上面代码的仿真结果。我们可以看到，输入 inp = "0011"（十进制数 3）；当输入 sel = 0（不移位）时，输出 outp = "00000011"（十进制数 3）；当输入 sel = 1（左移 1 位）时，输出 outp = "00000110"（十进制数 6）；当输入 sel = 2（左移 2 位）时，输出 outp = "00001100"（十进制数 12）。

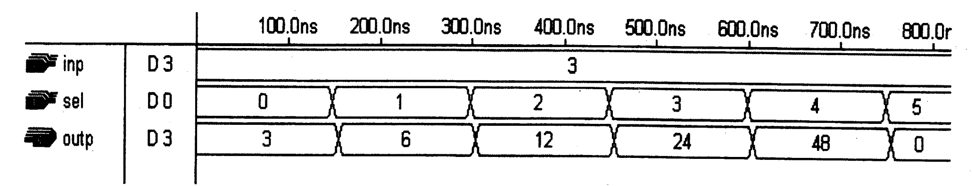


图 5.12 例 5.6 的仿真波形

5.5 块语句

VHDL 中存在两种类型的块(BLOCK)：简单块(simple BLOCK)和卫式块(guarded BLOCK)。

Simple BLOCK

Simple BLOCK 仅仅是一种对原有代码进行区域分割的方式。我们将一系列的并发描述语句放在一个 simple BLOCK 中，目的仅仅是为了增强整个代码的可读性和可维护性（对处理长代码很有帮助）。其语法结构如下：

```
label: BLOCK
  [declarative part]
BEGIN
  (concurrent statement)
END BLOCK label;
```

下面是采用 simple BLOCK 对代码进行归整后的一个构造体，可以看出整个构造体的结构变得更清晰了。

```
-------------------------------------------------
ARCHITECTURE example …
BEGIN
    ...
    block1: BLOCK
    BEGIN
        ...
    END BLOCK  block1;
    ...
    block2: BLOCK
    BEGIN
        ...
    END BLOCK  block2;
    ...
END example;
-------------------------------------------------
```

下面是一个完整定义的 BLOCK：

```
b1: BLOCK
    SIGNAL a: STD_LOGIC;
BEGIN
    a <= input_sig WHEN ena = '1' ELSE 'Z';
END BLOCK b1;
```

无论是 simple BLOCK 还是 guarded BLOCK，其内部都可以嵌套其他 BLOCK，相应的语法结构如下：

```
lable1: BLOCK
    [顶层 BLOCK 的声明部分]
BEGIN
    [顶层 BLOCK 的并发描述语句]
lable2: BLOCK
    [嵌套 BLOCK 的声明部分]
BEGIN
    (嵌套 BLOCK 的并发描述语句)
END BLOCK  label2;
    [顶层 BLOCK 的其他并发描述语句]
END BLOCK label1;
```

注意，我们将在本书的第二部分（系统设计部分）详细讨论如何将一段较长的代码根据需要进行划分。BLOCK 语句虽然也可以用于对代码进行分割，从而增强其可读性，但它的结构相对独立并且不调用 PACKAGE，COMPONENT，FUNCTION 或 PROCEDURE（这些将在第二部分进行学习），所以我们将其放在本章中讨论。

Guarded BLOCK

卫式块（guarded BLOCK）是一种特殊的 BLOCK，与 simple BLOCK 相比，它多了一个卫式表达式。只有当卫式表达式的值为真时，在 guarded BLOCK 中前面有关键词 GUARDED 的语句才能执行。

Guarded BLOCK 的语法结构如下：

```
label: BLOCK (卫式表达式)
   [声明部分]
BEGIN
   (卫式语句和其他并发描述语句)
END BLOCK  label;
```

前面讲到，BLOCK 中的语句是并发执行的，那是对大多数情况而言的。在下面的例子中可以看到，在卫式 BLOCK 内部，可以构造时序电路，只是这种应用并不常见。

例 5.7　用 guarded BLOCK 实现锁存器

下面给出了一个实现透明锁存器的例子。其中，clk = '1'（第 12 行）是卫式表达式，q <= GUARDED d（第 14 行）是卫式语句。可见，只有当 clk = '1'时，才执行 q <= d。

```
1  --------------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY latch IS
6     PORT ( d, clk: IN  STD_LOGIC;
7              q: OUT STD_LOGIC);
8  END  latch;
9  --------------------------------------------------
10 ARCHITECTURE latch OF latch IS
11 BEGIN
12   b1: BLOCK(clk = '1')
13   BEGIN
14      q <= GUARDED d;
```

```
15   END BLOCK b1;
16 END latch;
17 --------------------------------------------------
```

例 5.8 用 guarded BLOCK 实现 D 触发器

下面要设计一个带有同步复位、上升沿触发的 D 触发器。与上面的代码相似，clk EVENT AND clk = '1'（第 12 行）是卫式表达式，q <= GUARDED '0' WHEN rst='1'（第 14 行）是卫式语句。可见，只有卫式表达式为真且 rst 信号为高电平时，才执行 q <= '0'。

```
1  --------------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY dff IS
6     PORT ( d, clk, rst: IN  STD_LOGIC;
7             q: OUT STD_LOGIC);
8  END  dff;
9  --------------------------------------------------
10 ARCHITECTURE dff OF dff IS
11 BEGIN
12    b1: BLOCK(clk'EVENT AND clk = '1')
13    BEGIN
14       q <= GUARDED '0' WHEN rst = '1' ELSE d;
15    END BLOCK b1;
16 END dff;
17 --------------------------------------------------
```

5.6 习题

试编写并发代码来解答本节的问题。综合后对电路进行仿真，验证其正确性。

5.1 **通用多路复用器**。在例 5.1 和例 5.2 给出的多路复用器中，输入矢量的个数和每个输入矢量的位宽都是预先定义好的。图 P5.1 给出了一个通用的多路复用器，其中 n 代表有多少个输入矢量，m 代表每个输入矢量的位宽。如图所示，电路有 2^n 个输入（注意这里的 n 和 m 没有依赖关系）。试用 GENERIC 语句来指定 n 的值，并假设 m = 8。实现这个电路。

提示：输入应该定义成矢量数组，对此可以参考 3.5 节。

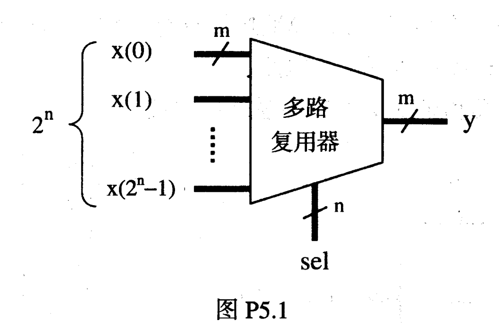


图 P5.1

5.2 **优先级编码器**。图 P5.2 是一个 7 级优先级编码器。如果输入矢量中出现多个'1'，那么电路将优先对最高位编码输出。"000"表示输入矢量中没有出现位'1'，不需要编码输出。按下面的要求实现该电路。

（1）只使用运算操作符；

（2）使用 WHEN/ELSE 语句。

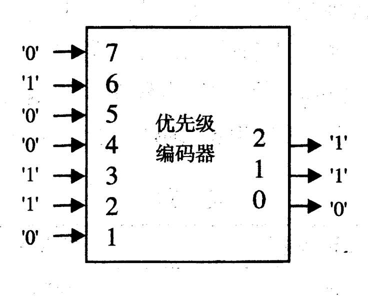


图 P5.2

5.3 **简单乘法器/除法器**。只使用并发描述代码实现图 P5.3 所示的乘法器/除法器。a 和 b 是 8 位整数类型的输入信号。x 和 y 是整数类型的输出信号，其中 x = a*b，y = a/2。

注意，要实现通用的定点除法器，可以参阅第 9 章。

5.4 **加法器**。只使用并发代码实现图 P5.4 中的 8 位无符号数加法器。

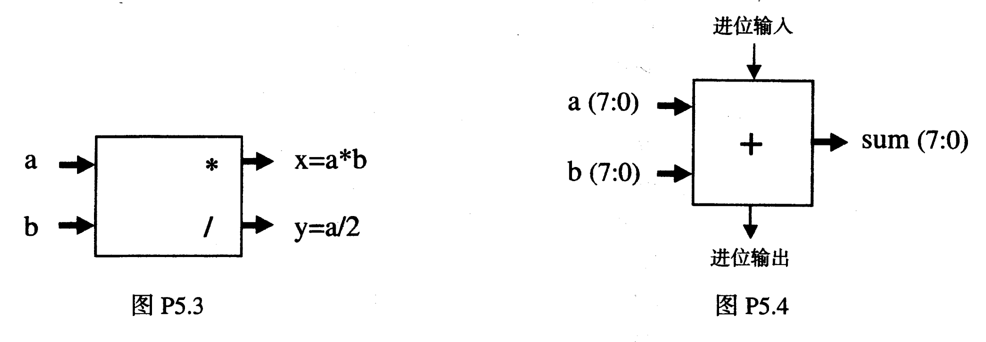
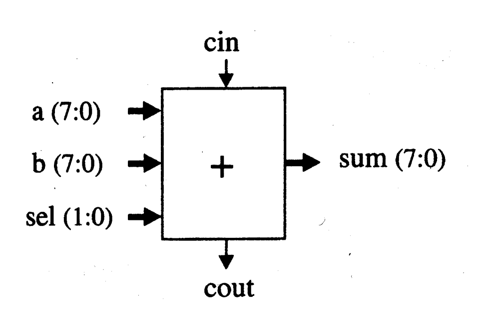


图 P5.3　　　　图 P5.4

5.5 **有符号数/无符号数加法器和减法器**。与习题 5.4 相比，图 P5.5 所示的电路增加了一个两位的

输入信号（sel），这样电路就可以有选择地执行有符号数/无符号数加法运算或减法运算（见真值表）。试编写 VHDL 代码实现这个电路。

提示：可以将得到的答案和第 9 章中的例题进行比较。



sel	运算
00	无符号加法
01	带符号加法
10	无符号减法
11	带符号减法

图 P5.5

5.6 二进制码–格雷码转换器。在数字系统中，我们用得最多的是二进制码。其最低位的权重是 2^0，权重按其位数以 2 的指数规律递增，最高位的权重是 2^{n-1}（n 是位宽）。另一方面，格雷码的相邻码字具有最小汉明距离。也就是说，相邻码字只有一位不同。当 n = 4 时，我们在表 P5.6 中列出了 4 位二进制码和格雷码的所有码字。试编写 VHDL 代码，实现从二进制码到格雷码的转换功能（位宽 n 应为 GENERIC 参数，以增加电路的通用性）。

表 P5.6

二进制编码	格雷码
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

5.7 **简易桶形移位寄存器**。图 P5.7 给出了一种简易的桶形移位寄存器的电路图。在这个电路中，输出或者是输入矢量（8 位）左移一位，或者等于输入。当矢量左移时，最低位填充'0'（见电路图的左下角），outp(0) = '0'；outp(i) = inp(i–1)，1≤i≤7。当矢量不移位时，输出就等于输入。试用并发代码来实现这个电路。

注意，对于通用的桶形移位寄存器（移动的位数可以是 0 至 n–1，n 是输入信号的位宽），参见第 9 章。

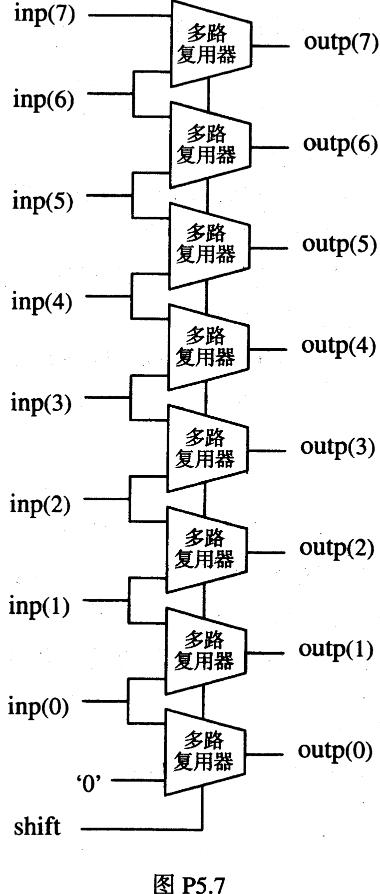


图 P5.7

5.8 **比较器**。编写 VHDL 代码实现对输入的两个 8 位矢量（a 和 b）进行比较操作的电路。选择信号 sel 决定是对无符号数比较（sel = '0'）还是对有符号数（sel = '1'）比较。电路有 3 个输出信号 x1，x2 和 x3，分别代表 a > b，a = b 和 a < b（见图 P5.8）。

可以将得到的答案和第 9 章中相应的例题进行比较。

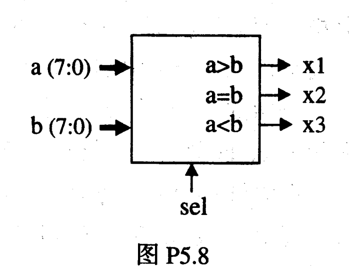


图 P5.8

第6章　顺序代码

在第 5 章中曾经提到，VDHL 本质上是一种并发执行的代码，但在 PROCESS，FUNCTION 和 PROCEDURE 内部的代码都是顺序执行的。当作为一个整体时，它们与外部的其他代码之间又都是并发执行的。

需要着重说明的一点是，顺序代码并非只能与时序逻辑相对应，同样可以用它们来实现组合逻辑电路。顺序代码也可以称为行为描述代码。

本章重点讨论的是顺序代码，也就是在 PROCESS，FUNCTION 和 PROCEDURE 内部出现的语句。这样的语句包括 IF，WAIT，CASE 和 LOOP 语句。

变量只能在顺序代码中使用，相对于信号而言，变量只能是局部的，所以它的值不能传递到 PROCESS，FUNCTION 和 PROCEDURE 的外部。

这里将重点讨论 PROCESS。FUNCTION 和 PROCEDURE 非常相似，并且主要用于系统级的设计，本书第二部分将讨论 FUNCTION 和 PROCEDURE 的应用。

6.1　进程

在 VHDL 中，进程（PROCESS）内部的语句是一种顺序描述语句，其内部经常使用 IF，WAIT，CASE 或 LOOP 语句。PROCESS 具有敏感信号列表（sensitivity list），或者使用 WAIT 语句进行执行条件的判断。PROCESS 必须包含在主代码段中，当敏感信号列表中的某个信号发生变化时（或者当 WAIT 语句的条件得到满足时），PROCESS 内部的代码就顺序执行一次。其语法结构如下：

```
[label: ]PROCESS (sensitivity list)
  [VARIABLE name: type [range][ := initial_value; ]]
BEGIN
  (顺序执行的代码)
END PROCESS [label];
```

其中的变量声明部分是可选的，如果要在 PROCESS 内部使用变量，则必须在关键字 BEGIN 之前的变量声明部分对其进行定义。变量的初始值是不可综合的，只是在仿真过程中使用。

上面的“lable”也是可选的，使用 lable 可以增强代码的可读性。可以使用关键字以外的任何名称对 lable 命名（参见附录 B）。

在设计一个同步电路时，必须对某些信号边沿的跳变进行监视（例如时钟信号 clock 的上升沿或下降沿）。通常使用 EVENT 属性来监视一个信号是否发生了变化（见 4.2 节）。例如，我们需要

监视 clk 信号的边沿跳变，那么当 clk 信号发生变化（上升沿或下降沿）时，clk'EVENT 将返回 TRUE。下面用一个例子来说明 EVENT 和 PROCESS 的用法。

例 6.1　带有异步复位端的 D 触发器

图 6.1 所示的 D 触发器是时序逻辑电路中最基本的单元。当输入的时钟信号发生变化（上升沿或下降沿）时，输出信号 q 的值将与当前的输入信号值相同。

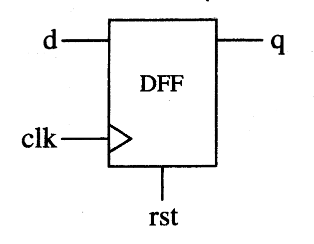


图 6.1　例 6.1 中带有异步复位端的 D 触发器

在下面给出的代码中，使用 IF 语句设计了一个带有异步复位端的 D 触发器（参见 6.3 节）。无论 clk 处于什么状态，只要 rst = '1'，就有输出 q = '0'（第 14 行~第 15 行）。否则，当 clk 信号的上升沿（第 16 行~第 17 行）出现时，输入的值将赋给输出（也就是说 q = d）。第 16 行使用了 EVENT 属性来检测时钟信号 clk 的变化。当敏感信号列表（clk 和 rst，第 12 行）中的某个信号发生变化时，PROCESS 中的语句就执行一次。图 6.2 给出了电路综合后的仿真波形，验证了电路的功能。

```
1  ---------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY dff IS
6     PORT ( d, clk, rst: IN  STD_LOGIC;
7            q: OUT STD_LOGIC);
8  END  dff;
9  ---------------------------------------------
10 ARCHITECTURE behavior OF dff IS
11 BEGIN
12    PROCESS (clk, rst)
13    BEGIN
14       IF (rst = '1') THEN
15          q <= '0';
16       ELSIF (clk'EVENT AND clk = '1') THEN
17          q <=  d;
18    END IF;
```

```
19      END PROCESS;
20  END behavior;
21  ---------------------------------------------------
```

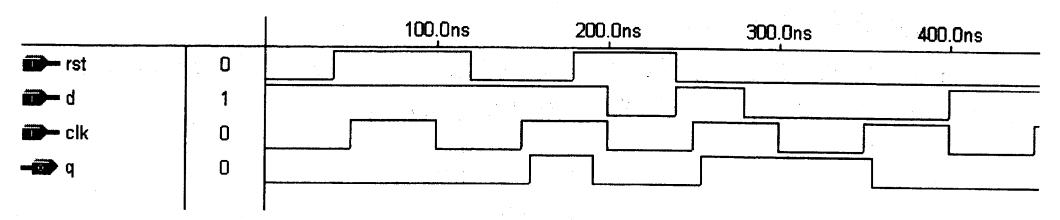


图 6.2 例 6.1 的仿真结果

6.2 信号和变量

尽管信号（SIGNAL）和变量（VARIABLE）是下一章要重点讨论的内容，但是在学习顺序执行代码时，读者有必要了解一些信号和变量的基本知识。

在 VHDL 中，有两种方法进行动态的数值传递：信号和变量。其中，信号可以在 PACKAGE，ENTITY 和 ARCHITECTURE 中声明，而变量只能在一段顺序描述代码的内部声明（比如在 PROCESS 内部进行信号声明）。因此，信号通常是全局的，而变量通常是局部的。

变量的值是无法直接传递到 PROCESS 外部的。如果需要进行变量值的传递，则必须把这个值赋给一个信号，然后由该信号将变量值传递到 PROCESS 外部。另一方面，赋予变量的值是立刻生效的，在此后的代码中，此变量将使用新的变量值。这一点和 PROCESS 中使用的信号不同，新的信号值通常只有在整个 PROCESS 运行完毕以后才开始生效。

最后，根据 4.1 节所讲的内容，可以知道信号的赋值操作符是“ <= ”（例如，sig <= 5），而变量的赋值操作符是“ := ”（例如，var := 5）。

6.3 IF 语句

前面曾经提到，IF，WAIT，CASE 和 LOOP 语句是用于顺序代码的。因此，它们只能用在 PROCESS，FUNCTION 和 PROCEDURE 中。

从设计人员的思维习惯出发，他们会更倾向于使用 IF 语句。使用 IF 语句对电路实现可能会带来负面的影响，因为 IF/ELSE 语句在综合时可能会产生不必要的优先级解码电路。目前的一些综合工具在处理这类语句时会对其结构进行优化，以避免占用过多的硬件资源。IF 语句的语法结构如下：

```
IF conditions THEN assignments;
ELSIF conditions THEN assignments;
...
```

```
ELSE assignments;
END IF;
```

例

```
IF (x<y) THEN temp := "11111111"; END IF
ELSIF (x=y AND w = '0') THEN temp := "11110000";
ELSE temp := (OTHERS => '0');
```

例 6.2　模 10 计数器#1

下面的代码实现了一个进行循环累加的模 10 计数器。顶层电路如图 6.3 所示。它的输入信号只有 clk，输出是位宽为 4 的信号 digit。代码中使用了 IF 语句。PROCESS 内部使用了变量 temp 来实现存储 4 位输出信号所需的 4 个 D 触发器。图 6.4 给出了电路综合后的仿真波形，验证了电路功能的正确性。

```
1  ---------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY counter IS
6     PORT ( clk: IN  STD_LOGIC;
7            digit: OUT INTEGER RANGE 0 TO 9);
8  END  counter;
9  ---------------------------------------------
10 ARCHITECTURE counter OF counter IS
11 BEGIN
12    count: PROCESS (clk)
13       VARIABLE temp: INTEGER RANGE 0 TO 10;
14    BEGIN
15       IF (clk'EVENT AND clk = '1') THEN
16          temp := temp+1;
17          IF (temp=10) THEN temp := 0;
18          END IF;
19       END IF;
20       digit <= temp;
21    END PROCESS count;
22 END counter;
23 ---------------------------------------------
```

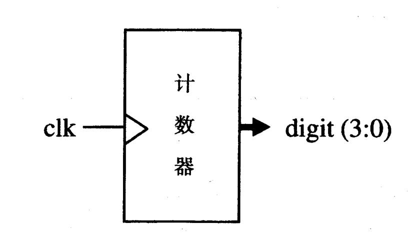


图 6.3　例 6.2 中的计数器

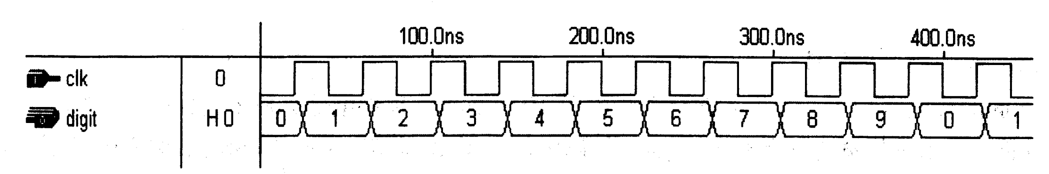


图 6.4　例 6.2 的仿真结果

需要注意的是，上面的电路中既没有复位输入端口，PROCESS 内部也没有对变量 temp 进行初始化（从而使 digit 没有确定的初始值）。因此，在实际电路刚启动时，temp 可以是任意值。如果 temp 的初始值小于 10，那么该电路会进行正常的累加计数；但如果这个值大于 10，那么计数器就需要消耗几个时钟周期，先累加计数到 15（"1111"），然后返回到 0，接着才能进行正常的计数操作。通常，在开始时浪费几个时钟周期并不是个大问题。要避免这个问题，可以将第 17 行的 temp = 10 改成 temp >= 10，这样可以避免启动时时钟周期的浪费，但显然会占用更多的硬件资源。如果想每次计数都从 0 开始，就必须引入复位信号（见例 6.7）。

在上面的代码中，为了在 temp 计数到 10 时对计数器复位，每次对 temp 累加时都将它与 10 进行比较，这是计数器设计中常用的一种方法。由于 10 是一个常数，编译器会产生一个相对来说比较简单的常数比较器电路。然而，如果不使用常数而使用一个可编程的参数，那么在电路实现时就会产生一个全比较器，这样会需要更多的逻辑资源。在这种情况下，更好的解决办法是在代码开头就把参数值赋给 temp，然后对 temp 进行递减计数操作，当递减到 0 的时候把参数值重新赋给 temp。这样，比较器是对 temp 和 0（常数）进行比较，就可以避免产生全比较器了。

例 6.3　移位寄存器

在图 6.5 所示的 4 位移位寄存器电路中，输出信号（q）滞后输入信号（d）4 个时钟周期。它带有一个异步复位端口（rst），当 rst = 1 时，所有的触发器全部复位。在本例题中，我们使用了 IF 语句。

```
1  --------------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY shiftreg IS
```

```
6     GENERIC (n: INTEGER := 4); --# of stages
7     PORT ( d, clk, rst: IN  STD_LOGIC;
8             q: OUT STD_LOGIC);
9   END  shiftreg;
10 --------------------------------------------------
11 ARCHITECTURE behavior OF shiftreg IS
12    SIGNAL internal: STD_LOGIC_VECTOR(n-1 DOWNTO 0);
13 BEGIN
14    PROCESS (clk, rst)
15    BEGIN
16       IF (rst = '1') THEN
17          internal <= (OTHERS => '0');
18       ELSIF (clk'EVENT AND clk = '1') THEN
19          internal <= d & internal(internal'LEFT DOWNTO 1);
20       END IF;
21    END PROCESS;
22    q <= internal(0);
23 END behavior;
24 --------------------------------------------------
```

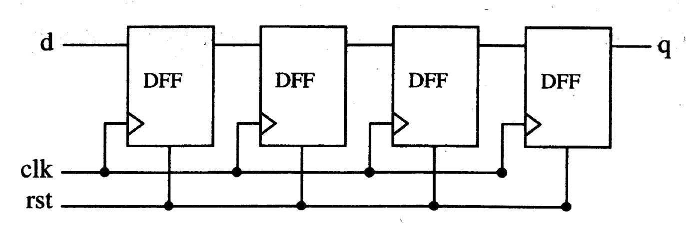


图 6.5　例 6.3 中的移位寄存器

上面代码的仿真结果如图 6.6 所示。我们可以看到，在 4 个上升沿之后，d 的值出现在 q 端。

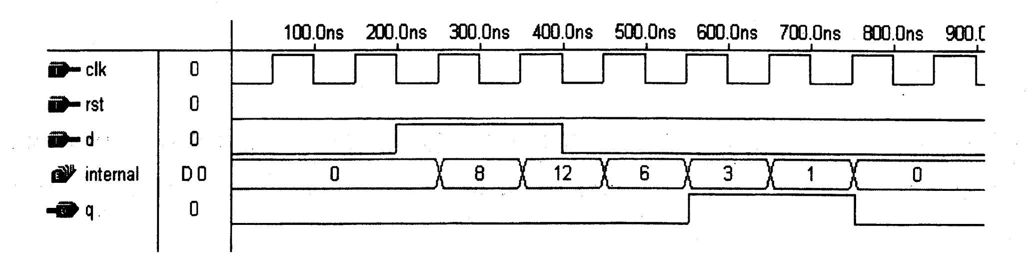


图 6.6　例 6.3 的仿真结果

6.4 WAIT 语句

WAIT 语句的用法和 IF 语句类似，但 WAIT 语句的形式更加多样。此外，不同于 IF，CASE 和 LOOP 语句，如果在 PORCESSS 中使用了 WAIT 语句，就不能再使用敏感信号列表了。WAIT 语句的使用有以下 3 种形式的语法结构：

```
WAIT UNTIL signal_condition;
```

含义：直到信号条件（signal_condition）满足时才进行后面的操作。

```
WAIT ON signal1[, signal2, …];
```

含义：当后面列出的信号中有一个发生变化时，开始进行后面的操作。

```
WAIT FOR time;
```

含义：等待 time 所确定的时间后开始进行后面的操作。

观察 WAIT UNTIL 语句可以发现，它后面只有一个信号条件表达式，因此更适合于实现同步电路。由于在使用 WAIT UNTIL 语句时，PROCESS 没有敏感信号列表，所以它必须是 PROCESS 中的第一条语句。当 WAIT UNTIL 语句的条件满足时，PROCESS 内部的代码就执行一遍。

例 带有同步复位端的 8 位寄存器

```
PROCESS          -- 没有敏感信号列表
BEGIN
   WAIT UNTIL (clk'EVENT AND clk = '1');
   IF (rst = '1') THEN
      output <= "00000000";
   ELSIF (clk'EVENT AND clk = '1') THEN
      output <= input;
   END IF;
END PROCESS;
```

WAIT ON 语句中可以出现多个信号。只要信号列表中的任何一个发生变化，PROCESS 内的代码就开始执行。在下面这个例子中，只要 rst 和 clk 信号发生变化，PROCESS 内部的代码就执行一次。

例 带异步复位端的8位寄存器

```
PROCESS
BEGIN
    WAIT ON clk, rst;
    IF (rst = '1') THEN
        output <= "00000000";
    ELSIF (clk'EVENT AND clk = '1') THEN
        output <= input;
    END IF;
END PROCESS ;
```

需要说明的是，WAIT FOR 语句只能用于仿真。例如，WAIT FOR 5ns 这样的代码是不可综合的。

例 6.4 带有异步复位端的 D 触发器#2

下面的代码实现了和例 6.1 中一样的 D 触发器（见图 6.1 和图 6.2）。然而，这里不仅使用了 IF 语句，还使用了 WAIT ON 语句。

```
1  ---------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  ENTITY dff IS
6     PORT (d, clk, rst: IN STD_LOGIC;
7           q: OUT STD_LOGIC);
8  END dff;
9  ---------------------------------------------------
10 ARCHITECTURE dff OF dff IS
11 BEGIN
12    PROCESS
13    BEGIN
14       WAIT ON rst, clk;
15       IF (rst = '1') THEN
16          q <= '0';
17       ELSIF (clk'EVENT AND clk = '1') THEN
18          q <= d;
19       END IF;
20    END PROCESS;
21 END dff;
22 ---------------------------------------------------
```

例 6.5　模 10 计数器#2

下面的代码实现了和例 6.2(见图 6.3 和图 6.4)相同的模 10 计数器。这里不仅使用了 IF 语句，还使用了 WAIT UNTIL 语句。

```
1  -----------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  -----------------------------------------------
5  ENTITY counter IS
6     PORT ( clk: IN  STD_LOGIC;
7             digit: OUT INTEGER RANGE 0 TO 9);
8  END  counter;
9  -----------------------------------------------
10 ARCHITECTURE counter OF counter IS
11 BEGIN
12    PROCESS  -- 没有敏感信号列表
13       VARIABLE temp: INTEGER RANGE 0 TO 10;
14    BEGIN
15       WAIT UNTIL(clk'EVENT AND clk = '1');
16       temp :=  temp+1;
17       IF (temp=10) THEN temp := 0;
18       END IF;
19       digit <= temp;
20    END PROCESS;
21 END counter;
22 -----------------------------------------------
```

6.5　CASE 语句

CASE 与 IF，LOOP 以及 WAIT 一样，也是一种专用于顺序代码中的语句。CASE 语句的语法结构如下：

```
CASE 表达式 IS
   WHEN 条件表达式 => 顺序执行语句;
   WHEN 条件表达式 => 顺序执行语句;
   …
END CASE;
```

例

```
CASE control IS
   WHEN"00" => x <= a; y <= b;
   WHEN"01" => x <= b; y <= c;
   WHEN OTHERS => x <= "0000"; y <= "zzzz";
END CASE;
```

CASE 语句和 WHEN 语句有些类似。由于需要考虑条件表达式所有可能出现的情况，所以关键词 OTHERS 非常有用，它代表了所有未列出的可能情况。另一个比较常用的关键词是 NULL（类似于 UNAFFECTED），它表示没有操作发生，例如 WHEN OTHERS => NULL。

•CASE 语句允许在每个测试条件下执行多个赋值操作，然而 WHEN 语句只允许执行一个赋值操作。

对于 5.3 节中所讲的 WHEN 语句，当需要对多种情况进行判断时，可以采用以下 3 种形式：

```
WHEN value                   -- 对单个值进行条件判断
WHEN value1 to value2        -- 对一个取值范围进行条件判断，仅适用于枚举类型
WHEN value1 | value2|…       -- 对多个可能出现的值进行判断，满足一个即可
```

例 6.6　带有异步复位端的 D 触发器#3

下面的代码实现了与例 6.1（见图 6.1 和图 6.2）相同的带有异步复位端的 D 触发器。然而，这里不仅使用了 IF 语句，还使用了 CASE 语句。注意，为了说明 CASE 语句的用法，我们有意引入了一些不必要的声明。

```
1  -------------------------------------------------
2  LIBRARY  ieee;                    -- 没有必要声明
3                                    -- 因为端口数据类型为 BIT
4  USE  ieee.std_logic_1164.all;     -- 而不是 STD_LOGIC
5
6  -------------------------------------------------
7  ENTITY dff IS
8     PORT ( d, clk, rst: IN  BIT;
9              q: OUT BIT);
10 END  dff;
11 -------------------------------------------------
12 ARCHITECTURE dff3 OF dff IS
13 BEGIN
14    PROCESS (clk, rst)
15    BEGIN
16       CASE rst IS
17          WHEN '1' =>  q <= '0';
```

```
18            WHEN '0' =>
19               IF (clk'EVENT AND clk = '1') THEN
20                  q <= d;
21               END IF;
22            WHEN OTHERS => NULL;
23         END CASE;
24      END PROCESS;
25 END dff3;
26 -------------------------------------------------
```

例 6.7 带 7 段数码显示的模 100 计数器

下面的代码实现了一个异步复位的模 100 累加计数器，此外它还可以将累加的 BCD（binary coded decimal，二进制编码的十进制数）值转换成 7 段数码显示（SSD：Seven Segment Display）电路驱动信号。SSD 的电路如图 6.7 所示。第 31 行~第 56 行使用了 CASE 语句来确定输出信号的值，这些值将传递给 SSD。注意，转换电路和 SSD 的连接关系是 xabcdefg，即最高位的值传递给小数点，而最低位的值传递给图中的段 g。

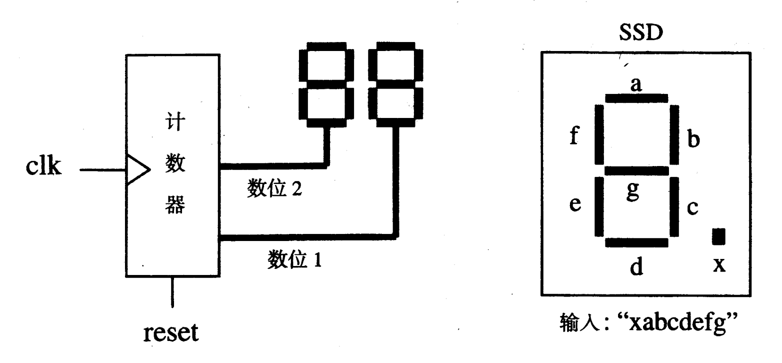


图 6.7 例 6.7 中的模 100 计数器

我们可以看到，这个电路是对例 6.2 中电路的直接扩展。不同之处在于，这里需要一个模 100 计数器，同时电路的输出是和 SSD 相连的。图 6.8 给出了电路的仿真结果。

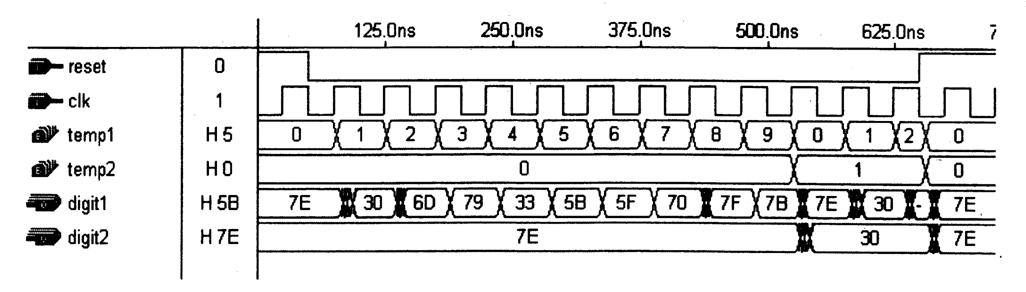


图 6.8 例 6.7 的仿真结果

```
1  ------------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY counter IS
6     PORT ( clk, reset: IN  STD_LOGIC;
7            digit1, digit2: OUT STD_LOGIC_VECTOR(6 DOWNTO 0));
8  END  counter;
9  ------------------------------------------------
10 ARCHITECTURE counter OF counter IS
11 BEGIN
12    PROCESS (clk, reset)
13       VARIABLE temp1: INTEGER RANGE 0 TO 10;
14       VARIABLE temp2: INTEGER RANGE 0 TO 10;
15    BEGIN
16    --counter: -----
17     IF (reset = '1') THEN
18        temp1 :=  0;
19        temp2 :=  0;
20     ELSIF (clk'EVENT AND clk = '1') THEN
21        temp1 := temp1+1;
22        IF (temp1=10) THEN
23           temp1 :=  0;
24           temp2 :=  temp2+1;
25           IF (temp2=10) THEN
26              temp2 := 0;
27           END IF;
28        END IF;
29     END IF;
30     ---BCD to SSD conversion: ---
31     CASE temp1 IS
32        WHEN 0 => digit1 <= "1111110"; --7E
33        WHEN 1 => digit1 <= "0110000"; --30
34        WHEN 2 => digit1 <= "1101101"; --6D
35        WHEN 3 => digit1 <= "1111001"; --79
36        WHEN 4 => digit1 <= "0110011"; --33
37        WHEN 5 => digit1 <= "1011011"; --5B
38        WHEN 6 => digit1 <= "1011111"; --5F
39        WHEN 7 => digit1 <= "1110000"; --70
```

```
40          WHEN 8 => digit1 <= "1111111"; --7F
41          WHEN 9 => digit1 <= "1111011"; --7B
42          WHEN OTHERS => NULL;
43       END CASE;
44       CASE temp2 IS
45          WHEN 0 => digit1 <= "1111110"; --7E
46          WHEN 1 => digit1 <= "0110000"; --30
47          WHEN 2 => digit1 <= "1101101"; --6D
48          WHEN 3 => digit1 <= "1111001"; --79
49          WHEN 4 => digit1 <= "0110011"; --33
50          WHEN 5 => digit1 <= "1011011"; --5B
51          WHEN 6 => digit1 <= "1011111"; --5F
52          WHEN 7 => digit1 <= "1110000"; --70
53          WHEN 8 => digit1 <= "1111111"; --7F
54          WHEN 9 => digit1 <= "1111011"; --7B
55          WHEN OTHERS => NULL;
56       END CASE;
57     END PROCESS;
58 END counter;
59 --------------------------------------------------
```

在上面的代码中使用了两次 CASE 语句，两次调用之间除了所判断的变量（temp1 和 temp2）不同以外，整个代码段完全相同。在这种情况下如何进行设计简化呢？在第二部分中将学习如何将常用代码加入到用户自定义的库中共享使用，从而避免上面出现的代码重复的情况。

6.6 LOOP 语句

当一段代码需要多次重复执行时，LOOP 语句就显得非常有效。与 IF，WAIT 和 CASE 语句一样，LOOP 语句也是顺序描述语句，所以只能用于 PROCESS，FUNCTION 和 PROCEDURE 中。

LOOP 语句有多种形式，其语法结构如下：

FOR/LOOP：循环固定次数

```
[label:]  FOR 循环变量 IN 范围 LOOP
  (顺序描述语句)
END LOOP [label];
```

WHILE/LOOP：循环执行直到某个条件不再满足

```
[label:] WHILE 条件表达式 LOOP
   (顺序描述语句)
END LOOP [label];
```

EXIT：结束整个循环操作

```
[label:] EXIT  [label]  [WHEN 条件表达式];
```

NEXT：跳出本次循环

```
[label:] NEXT [loop_label] [WHEN 条件表达式];
```

例　FOR/LOOP

```
FOR i IN 0 TO 5 LOOP
   x(i) <= enable AND w(i+2);
   y(0, i) <= w(i);
END LOOP;
```

在上面的代码中，LOOP 循环会无条件执行，直到 i 等于 5（即执行了 6 次）。

需要注意的是（和第 5 章中对 GENERATE 语句的限制有些相似），FOR/LOOP 语句中的上下界必须是静态值。因此，对于语句 FOR i IN 0 TO choice LOOP 来说，当 choice 是可变的输入参数时，代码通常是不可综合的。

例　WHILE/LOOP：当满足 i<10 时，LOOP 循环会一直执行。

```
WHILE (i<10) LOOP
   WAIT UNTIL clk'EVENT AND clk = '1';
   (其他语句)
END LOOP;
```

例　在下面的代码中，EXIT 不是表示跳出当前的循环，而是表示跳出整个循环体（也就是说，尽管 i 还没有达到跳出循环的边界值，LOOP 循环也要结束）。在这种情况下，只要矢量 data 的第 i 位的位等于零，循环就会结束。

```
FOR i IN 0 TO data'RANGE LOOP
   CASE data(i) IS
      WHEN '0' => count := count+1;
      WHEN OTHERS => EXIT;
   END CASE;
END LOOP;
```

例 在下面的代码中，当 i = skip 时，NEXT 语句会使代码执行跳出本次循环。

```
FOR i IN 0 TO 15 LOOP
   NEXT WHEN i=skip;  -- 跳到下一次循环
      (…)
END LOOP;
```

下面给出了一些使用 LOOP 语句的完整的设计实例。

例 6.8 逐级进位加法器

图 6.9 是一个逐级进位加法器电路，其中 a 和 b 是输入矢量，且都是 8 位无符号数，cin 是输入的进位位，s 是输出的和，cout 是输出的进位位。底层电路结构图说明了进位位的传递方式。

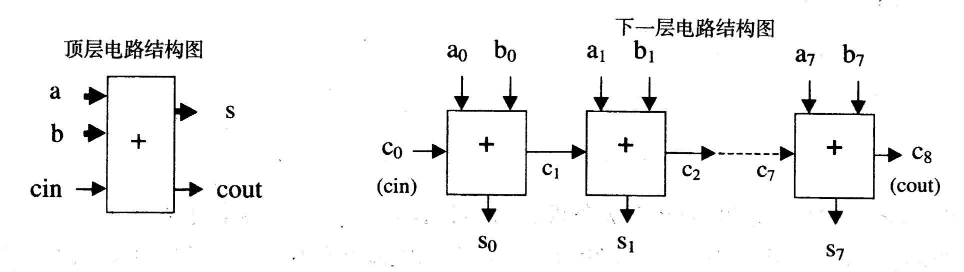


图 6.9 例 6.8 中的 8 位逐级进位加法器

从图 6.9 可以看出，8 位逐级进位加法器是由 8 个 1 位全加器电路（见 1.4 节）级联构成的。因此，8 位逐级进位加法器的输出可以按照下面的公式计算：

$s_j = a_j \text{ XOR } b_j \text{ XOR } c_j$

$c_{j+1} = (a_j \text{ AND } b_j) \text{ OR } (a_j \text{ AND } c_j) \text{ OR } (b_j \text{ AND } c_j)$

下面给出了两段实现逐级进位加法器的 VHDL 代码。第一段是通用的，它的输入矢量的宽度由 GENERIC 确定；第二段位宽固定为 8。此外，在前面一段代码中将输入定义为矢量，并使用了 FOR/LOOP 语句；在第二段代码中，输入为整数类型的信号，使用的是 IF 语句。它们的仿真波形如图 6.10 所示。在第 9 章中有更多关于加法器的内容。

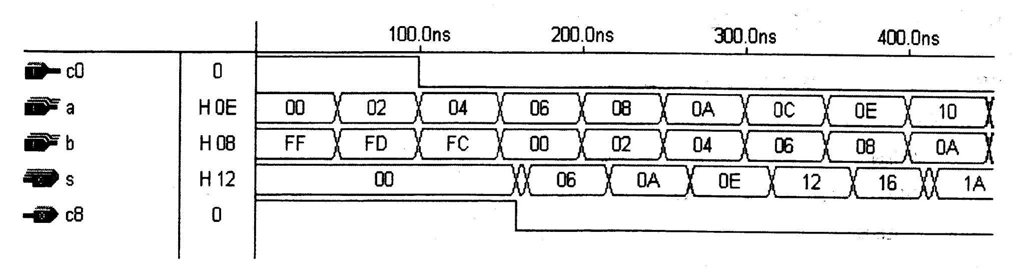


图 6.10 例 6.8 的仿真结果

```
1  -----------方案1: Generic, with vectors----------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY adder IS
6     GENERIC (length: INTEGER := 8);
7     PORT ( a, b: IN  STD_LOGIC_VECTOR(length-1 DOWNTO 0);
8            cin: IN STD_LOGIC;
9            S: OUT STD_LOGIC_VECTOR(length-1 DOWNTO 0);
10           cout: OUT STD_LOGIC);
11  END  adder;
12 --------------------------------------------------
13 ARCHITECTURE adder OF adder IS
14 BEGIN
15    PROCESS (a, b, cin)
16       VARIABLE carry: STD_LOGIC_VECTOR(length DOWNTO 0);
17    BEGIN
18       carry(0) := cin;
19       FOR i IN 0 TO length-1 LOOP
20          s(i) <= a(i) XOR b(i) XOR carry(i);
21          carry(i+1) :=  (a(i) AND b(i)) OR (a(i) AND
22                         carry(i)) OR (b(i) AND carry(i)) ;
23       END LOOP;
24       cout <= carry(length);
25    END PROCESS;
26 END adder;
27 --------------------------------------------------
```

```
1  --------方案2: non-generic, with INTEGERS---------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY adder IS
6     PORT ( a, b: IN  INTEGER RANGE 0 TO 255;
7            c0: IN STD_LOGIC;
9            S: OUT INTEGER RANGE 0 TO 255;
9            c8: OUT STD_LOGIC);
10 END  adder;
11 --------------------------------------------------
12 ARCHITECTURE adder OF adder IS
```

```
13 BEGIN
14    PROCESS (a, b, c0)
15       VARIABLE temp: INTEGER RANGE 0 TO 511;
16    BEGIN
17       IF (c0 = '1') THEN temp := 1;
18       ELSE temp := 0;
19       END IF;
20       temp := a+b+temp;
21       IF (temp>255) THEN
22          c8 <= '1';
23          temp := temp - 256;
24       ELSE c8 <= '0';
25       END IF;
26       s <= temp;
27    END PROCESS;
28 END adder;
29 ---------------------------------------------------
```

例 6.9　简易桶形移位寄存器

图 6.11 给出了一个简单桶形移位寄存器的电路图。在这个例子中，输出值或者是输入矢量左移一位得到的结果，或者就等于输入矢量（不进行移位）。当输入矢量左移一位（shift = 1）时，outp(0) = '0'，outp(i) = inp(i–1)，1≤i≤7。移位后空出的最低位将填充'0'（如电路图的左下角所示）。当输入矢量不移位时，输出与输入相同。

下面给出了完整的 VHDL 代码，其中展示了 FOR/LOOP 语句的使用方法。仿真结果如图 6.12 所示。

注意，第 9 章将给出完整的桶形移位寄存器（shift 的取值可以从 0 到 n–1，其中 n 是输入矢量的位宽）。

```
1  ---------------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  ENTITY barrel IS
6     GENERIC (n: INTEGER := 8) ;
7     PORT ( inp: IN  STD_LOGIC_VECTOR(n-1 DOWNTO 0);
8            shift: IN  INTEGER RANGE 0 TO 1;
9            outp: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
10 END  barrel;
11 ---------------------------------------------------
```

```
12 ARCHITECTURE RTL OF barrel IS
13 BEGIN
14    PROCESS (inp, shift)
15    BEGIN
16       IF (shift=0) THEN
17          outp <= inp;
18       ELSE
19          outp(0) <= '0';
20          FOR i IN 1 TO inp'HIGH LOOP
21            outp(i) <= inp(i-1);
22          END LOOP;
23       END IF;
24    END PROCESS;
25 END RTL;
26 ---------------------------------------------------
```

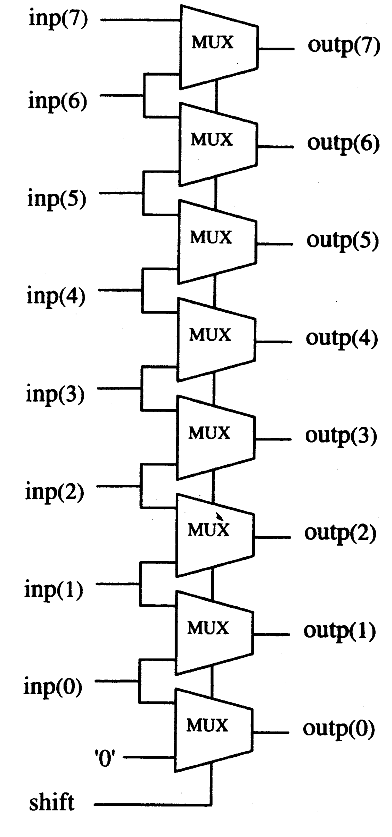


图 6.11　例 6.9 中的简易桶形移位寄存器

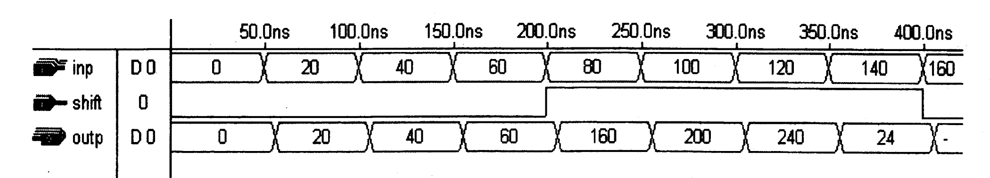


图 6.12　例 6.9 的仿真结果

例 6.10　连'0'检测器

在下面的设计中将对输入矢量中连续出现的零的个数进行统计（从矢量左端开始统计）。这里使用了 LOOP/EXIT 语句。需要注意的是，EXIT 语句表示跳出整个循环体，而不是跳出本次循环（也就是说，尽管 i 还没有达到跳出循环体的边界值，LOOP 循环还是会被强行终止）。在检测和统计开始后，只要在输入矢量中发现了'1'，LOOP 就会结束，因此整个电路检测的是输入矢量中从左端开始第一个'1'之前连续'0'的个数。

```
1  --------------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY LeadingZeros IS
6     PORT ( data: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7            zeros: OUT INTEGER RANGE 0 TO 8);
8  END  LeadingZeros;
9  --------------------------------------------------
10 ARCHITECTURE behavior OF LeadingZeros IS
11 BEGIN
12    PROCESS (data)
13       VARIABLE count: INTEGER RANGE 0 TO 8;
14    BEGIN
15       count := 0;
16       FOR i IN data'RANGE LOOP
17          CASE data(i) IS
18             WHEN '0' =>  count := count+1;
19             WHEN OTHERS => EXIT;
20          END CASE;
21       END LOOP;
22       zeros <= count;
23    END PROCESS;
24 END behavior;
25 --------------------------------------------------
```

如图 6.13 所示，仿真结果证明了电路功能是正确的。当数据为"00000000"（十进制的 0）时，电路检测到 8 个'0'；当数据为"00000001"（十进制的 1）时，电路检测到 7 个'0'；类似地，可以看出后面的结果也是正确的。

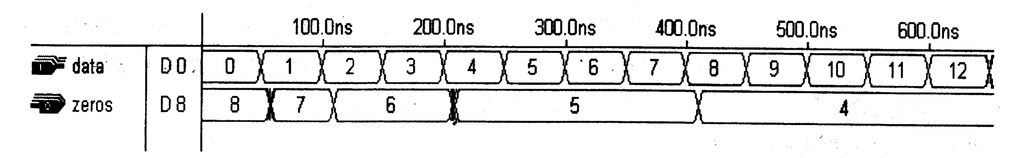


图 6.13 例 6.10 的仿真结果

6.7 CASE 语句和 IF 语句的比较

尽管在原则上 IF/ELSE 语句中 ELSE 的出现可能会造成综合后的电路中出现优先级解码器（CASE 语句不会出现这种情况），但实际综合时由于综合工具的优化功能往往不会出现这种情况。例如，我们使用 IF 语句来实现一个纯组合逻辑电路时，IF 语句会综合成一个多路复用器而不是优先级解码器。因此，用 IF 语句和 CASE 语句编写的代码在综合、优化后最终生成的电路结构通常是一样的。

例 下面的代码综合后可以得到结构相同的多路复用器。

```
---------------with IF: ------------------
IF (sel="00") THEN x <= a;
ELSIF (sel="01") THEN x <= b;
ELSIF (sel="10") THEN x <= c;
ELSE  x <= d;
--------------with CASE: -----------------
CASE sel IS
   WHEN "00" => x <= a;
   WHEN "01" => x <= b;
   WHEN "10" => x <= c;
   WHEN OTHERS => x <= d;
END CASE;
------------------------------------------
```

6.8 CASE 语句和 WHEN 语句的比较

CASE 语句和 WHEN 语句非常相似，不同之处在于 CASE 语句是顺序执行的，WHEN 语句是并发执行的。表 6.1 对 CASE 语句和 WHEN 语句的异同进行了总结。

表 6.1 WHEN 和 CASE 的比较

	WHEN	CASE
代码类型	并发代码	顺序代码
用法	在 PROCESSES，FUNCTIONS 和 PROCEDURES 外部使用	在 PROCESSES，FUNCTIONS 和 PROCEDURES 内部使用
必须列出所有可能的组合	是（对于 WITH/SELECT/WHEN 来说）	是
每个判断分支允许的最大赋值操作数量	1	任意
没有操作动作时使用的关键字	UNAFFECTED	NULL

例 从实现的功能上看，下面的两段代码是等效的。

```
------------------with WHEN: --------------------
WITH sel SELECT
  x <= a WHEN "000";
       b WHEN "001";
       c WHEN "010";
     UNAFFECTED WHEN OTHERS;
------------------with CASE: --------------------
CASE sel IS
   WHEN "000" => x <= a;
   WHEN "001" => x <= b;
   WHEN "100" => x <= c;
   WHEN OTHERS => NULL;
END CASE;
-------------------------------------------------
```

6.9 同步时序电路中的时钟问题

在代码中，如果在参考时钟的两个边沿（上升沿和下降沿）都可以触发对同一个信号的赋值操作，那么这样的代码通常是不可综合的。特别是采用 CPLD（见附录 A）这类可编程器件进行电路实现时更是如此。如果出现这种情况，编译器就会给出“信号值在时钟边沿不能保持”等提示。

下面，我们以一个计数器为例对此进行分析。这个计数器在每个时钟边沿都进行计数操作。下面给出了其中一种实现方法：

```
PROCESS (clk)
BEGIN
   IF (clk'EVENT AND clk = '1')  THEN
       counter <= counter+1;
```

```
    ELSIF (clk'EVENT AND clk = '0') THEN
        counter <= counter+1;
    END IF;
    …
END PROCESS;
```

在上面的代码进行编译时，除了给出前面所说的警告提示外，编译器还会提示信号 counter 是“多驱动”的，然后终止编译过程。

非常重要的另一点是，EVENT 属性必须和某个测试条件关联起来。例如语句 IF (clk'EVENT AND clk = '1')是正确的，但语句 IF (clk'EVENT)存在二义性。有些编译器认为 AND clk = '1'是默认的，而有些编译器会给出“时钟不稳定”的警告。下面仍以一个计数器为例，这个计数器在每个时钟沿都进行计数操作。有人编写了这样一段代码：

```
PROCESS (clk)
BEGIN
    IF (clk'EVENT) THEN
        counter := counter+1;
    END IF;
    …
END PROCESS;
```

尽管我们希望在每个时钟边沿出现时 PROCESS 都执行一次，计数器在每个时钟沿都进行计数操作，但基于上面提到的原因，这样的情况是不会发生的。如果编译器假定了一个默认值，那么只有在时钟上升沿出现时进行了计数操作，这和设计初衷显然是不符合的。如果编译器没有假定一个默认值，那么它会给出错误提示信息并停止编译。

最后，如果一个信号出现在敏感信号列表中，但没有在 PROCESS 内部的任何语句中出现，通常编译器会把它忽略掉。我们仍以上面提到的双边沿计数器为例，有下面这样一段代码：

```
PROCESS (clk)
BEGIN
    counter := counter+1;
    …
END PROCESS;
```

上面的代码的本意是希望计数器在时钟的上升沿和下降沿都进行计数操作，然而编译器会提示“忽略了不必要的信号 clk”。

例 不同于上面给出的代码，下面的一段代码包括两个 PROCESS，它可以顺利地通过编译和综合。注意，在两个 PROCESS 中使用了不同的信号。

```
-------------------------------------------------
PROCESS (clk)
BEGIN
   IF (clk'EVENT AND clk = '1') THEN
       x <= d;
   END IF;
END PROCESS;
-------------------------------------------------
PROCESS (clk)
BEGIN
    IF (clk'EVENT AND clk = '0') THEN
        y <= d;
    END IF;
END PROCESS;
-------------------------------------------------
```

现在读者已经知道了可以使用哪些语句，不能使用哪些语句。下面尝试解答习题 6.1。

例 6.11　RAM

下面给出了一段实现 RAM（random access memory）的顺序描述代码，其中使用了较多的 IF 语句。

从图 6.14（a）中可以看到，该电路具有输入数据总线（data_in）、输出数据总线（data_out）、地址总线（addr）、外加时钟信号（clk）和写使能信号（wr_ena）。

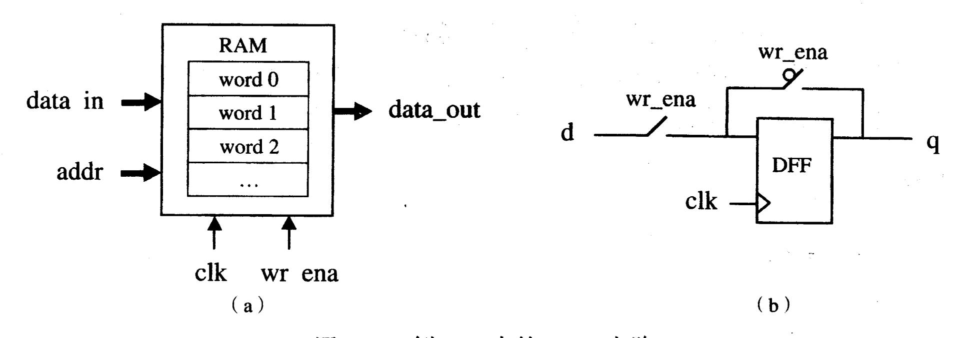


图 6.14　例 6.11 中的 RAM 电路

在 RAM 电路中，一旦 wr_ena 有效，那么在其后的下一个时钟上升沿出现时，data_in 上的数据将被写入地址总线所指定的位置上。在输出方向上，data_out 上显示的始终是当前 addr 所指定的 RAM 空间中的内容。

如果从寄存器的角度来看这个电路，其电路结构如图 6.14（b）所示。当 wr_ena 信号为低电平时，q 直接连接到触发器的输入端，且端口 d 断开，此时不会有新数据被写入到存储器中。当

wr_ena 信号为高时，d 连接到触发器的输入端，在下一个时钟上升沿到来时，d 将被写入到存储器中，并覆盖掉前一次写入的值。

下面给出了实现图 6.14 所示电路的 VHDL 代码。RAM 的深度为 16，位宽为 8。注意，由于使用了 GENERIC 语句，下面的代码具有通用性。图 6.15 给出了电路综合后的仿真结果。

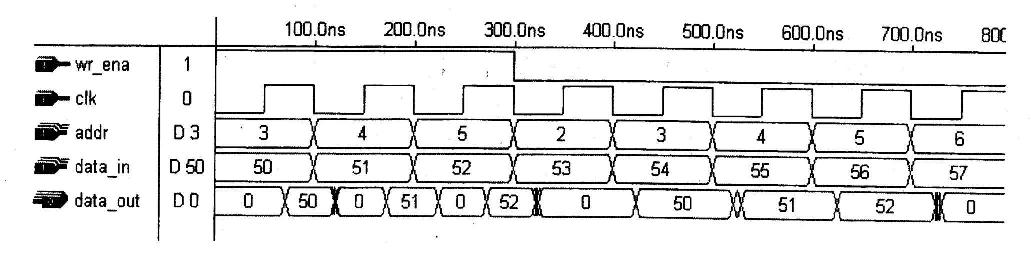


图 6.15　例 6.11 的仿真结果

9.10 节将给出其他存储器的实现方法。

```
1  ---------------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  ENTITY ram IS
6  GENERIC ( bits: INTEGER := 8;       -- # of bits per word
7              words: INTEGER := 16);  -- # of words in the memory
8     PORT ( wr_ena, clk: IN  STD_LOGIC;
9              addr: IN  INTEGER RANGE 0 TO words-1;
10             data_in: IN STD_LOGIC_VECTOR(bits-1 DOWNTO 0);
11             data_out: OUT STD_LOGIC_VECTOR(bits-1 DOWNTO 0));
12 END  ram;
13 ---------------------------------------------------
14 ARCHITECTURE ram OF ram IS
15    TYPE vector_array IS ARRAY(0 TO words-1) OF
16       STD_LOGIC_VECTOR(bits-1 DOWNTO 0);
17    SIGNAL memory: vector_array;
18 BEGIN
19    PROCESS (clk, wr_ena)
20    BEGIN
21       IF (wr_ena = '1') THEN
22          IF (clk'EVENT AND  clk = '1') THEN
23             memory(addr) <= data_in;
24          END IF;
```

```
25        END IF;
26     END PROCESS;
27     data_out <= memory(addr);
28  END ram;
29  --------------------------------------------------
```

6.10 使用顺序代码设计组合逻辑电路

我们已经知道，用顺序代码可以实现时序逻辑电路和组合逻辑电路。对于前者，代码综合后会生成寄存器。如果希望所编写的代码综合出一个组合逻辑电路，那么该组合逻辑电路的真值表必须在代码中被完整地反映出来。为了达到这一要求，必须遵守下面的原则：

原则 1：确保在 PROCESS 中用到的所有输入信号都出现在敏感信号列表中。

原则 2：确保考虑了输入/输出信号的所有可能组合，也就是说，电路的真值表必须在代码中完整地反映出来（实际上对顺序代码和并发代码都有这一要求）。

如果没有遵守原则 1，那么编译器通常会提示"某个输入信号没有在敏感信号列表中列出"，但编译器仍会继续执行，好像这个输入信号已经出现在敏感信号列表中一样。尽管这样做对设计不会带来多大影响，但是对一个优秀的设计而言，遵守原则 1 是必要的。

如果不遵守原则 2，后果更严重一些。因为没有考虑到输出信号的所有情况，为了保存以前的值，在综合时会产生一个锁存器。下面这个例子解释了这种情况。

例 6.12 会错误生成锁存器的组合逻辑设计

让我们分析图 6.16 所示的电路。如图 6.16（b）给出的真值表所示，x 是多路复用器的输出端，其输出值等于 sel 所选择的一个输入端的值。另一方面，当 sel = "00"时，y = '0'；当 sel = "01"时，y = '1'。

我们希望采用组合逻辑电路实现该真值表描述的逻辑功能。需要注意的是，图 6.16（b）的真值表中没有给出当 sel 分别为"10"和"11"时 y 的输出值。下面给出的是与如图 6.16（b）所示真值表相对应的代码。

```
1  --------------------------------------------------
2  LIBRARY  ieee;
3  USE  ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY example IS
6     PORT (a, b, c, d: IN  STD_LOGIC;
7       sel: IN  INTEGER RANGE 0 TO 3;
8       x, y: OUT STD_LOGIC);
9  END  example;
```

```
10 ---------------------------------------------------
11 ARCHITECTURE example OF example IS
12 BEGIN
13    PROCESS (a, b, c, d, sel)
14    BEGIN
15       IF (sel=0) THEN
16          x <= a;
17          y <= '0';
18       ELSIF (sel=1) THEN
19          x <= b;
20          y <= '1';
21       ELSIF (sel=2) THEN
22          x <= c;
23       ELSE
24          x <= d;
25       END IF;
26    END PROCESS;
27 END example;
28 ---------------------------------------------------
```

a b c d x y sel (1:0)

（a）顶层电路

sel	x	y
00	a	0
01	b	1
10	c	
11	d	

（b）预先给出的真值表

sel	x	y
00	a	0
01	b	1
10	c	y
11	d	y

（c）实际实现的真值表

sel	x	y
00	a	0
01	b	1
10	c	X
11	d	X

（d）正确实现的真值表

图 6.16 例 6.12 中的电路

上面的代码经过编译后，报告文件中显示并没有生成锁存器（和期望的一样）。然而，当观察图 6.17 给出的仿真结果时，会发现 y 的波形存在异常。对于同样的输入，y 出现了两种不同的输出值（如图所示，当两次出现 sel = 3 时，y 的输出结果不同）。这表明编译时的确生成了某种记忆单元。事实上，如果观察由 Quartus II（见附录 D）综合得到的结果，会发现 y = (sel(0) AND sel(1)) OR (sel(0) AND y) OR (sel(1) AND y)，这里面隐含着一个由与门和或门组成的锁存器，它造成了如图 6.16（c）所示的真值表。

为了避免产生锁存器，需要使用图 6.16（d）所示的真值表，其中使用了'X'表示“不确定”或“不需要考虑”。此时，需要在上面代码的第 22 行和第 24 行之间加入 y <= 'X' 这条语句。这样，经过综合以后，输出结果可以简单地表示为 y = sel(0)。

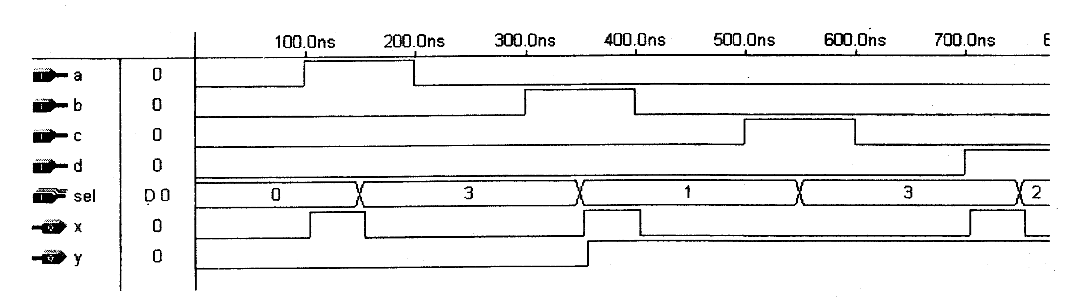


图 6.17　例 6.12 的仿真结果

6.11　习题

和上面的例子一样，下面给出的习题同样是为了说明顺序代码的结构，即如何在 PROCESS 语句中使用 IF，WAIT，CASE 和 LOOP 语句。在开始下面的习题之前，如果读者想了解关于信号（SIGNAL）和变量（VARIABLE）的更多知识，可以先参阅第 7 章，然后再来做本章的习题。在本章的习题中，只允许使用顺序代码来完成设计。由于使用顺序代码可以实现组合逻辑电路和时序逻辑电路，试分析下面每个习题中实现的究竟是组合逻辑电路还是时序逻辑电路。

6.1　边沿计数器。参见图 P6.1，设计这样一个电路，它能够统计时钟上升沿和下降沿数目之和。

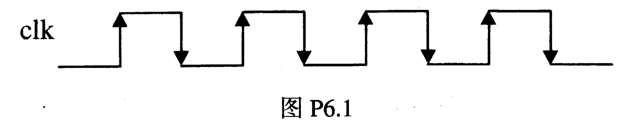


图 P6.1

6.2　移位寄存器。设计一个如图 P6.2 所示的由 4 个 D 触发器组成的移位寄存器。试给出和例 6.3 不同的实现方法。

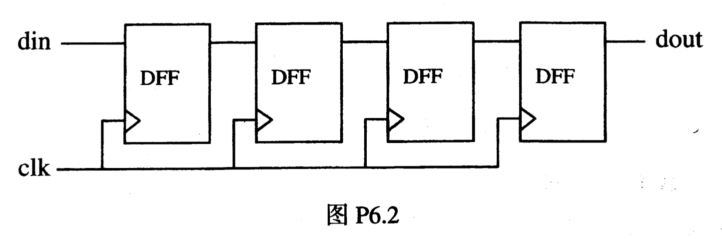


图 P6.2

6.3　优先级编码器。图 P6.3 给出了和习题 5.2 中一样的优先级编码器。编码输出结果将指出输入矢量中从高位开始第一个'1'出现的位置。输出值"000"表示输入矢量中没有出现位'1'。试用顺序代码写出实现该电路的两种方法。

（1）使用 IF 语句；

（2）使用 CASE 语句。

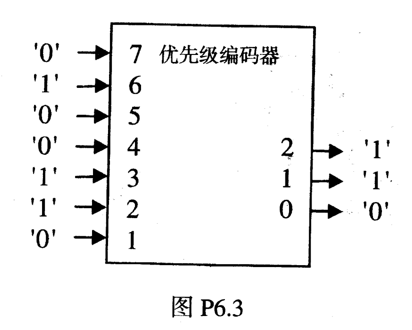


图 P6.3

6.4 **通用分频器**。试设计一个通用的 n 分频电路(见图 P6.4),其中 n 为整数。提示:使用 GENERIC 语句。

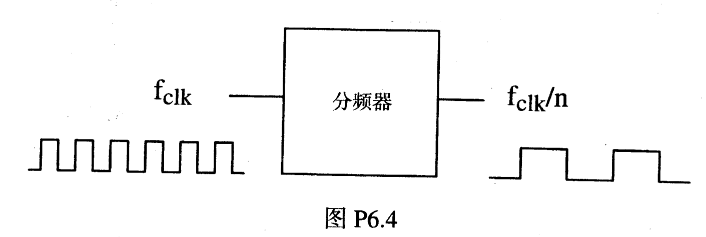


图 P6.4

6.5 **倍频器**。和习题 6.4 相反,试设计一个 n 倍频器,同样 n 为整数。该电路能够实现吗?

6.6 **计时器#1**。设计一个计时器,它能从 0 秒计时到 9 分 59 秒(见图 P6.6),当前时间可以在 SSD(7 段数码显示器)上显示。要求电路具有启动、停止和复位按钮,时钟频率为 1 Hz。

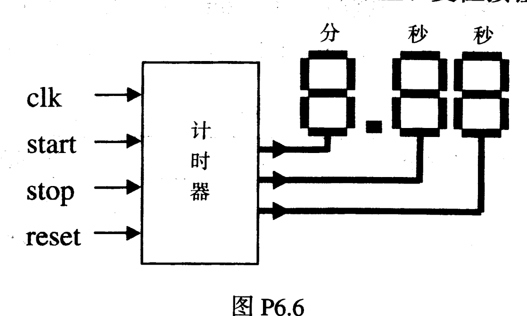


图 P6.6

6.7 **计时器#2**。重新设计习题 6.6 中的计时器,假设只能使用一个按钮,轮流来实现启动和停止功能。如果连续按着这个按钮超过 2 s,电路将复位。实现该计时器(见图 P6.7),时钟频率同样为 1 Hz。

6.8 **奇偶校验检测器**。图 P6.8 给出了一种奇偶校验检测器的顶层电路,输入矢量有 8 位。当输入矢量中'1'的个数是偶数时输出为'0',否则输出为'1'。试用顺序描述代码来实现这个电路,如果可能,试给出多种实现方法。

6.9 **'1'计数器**。试设计一个电路,它可以计算输入矢量中的'1'的个数(见表 P6.9)。编写 VHDL 代码,综合后对其仿真。

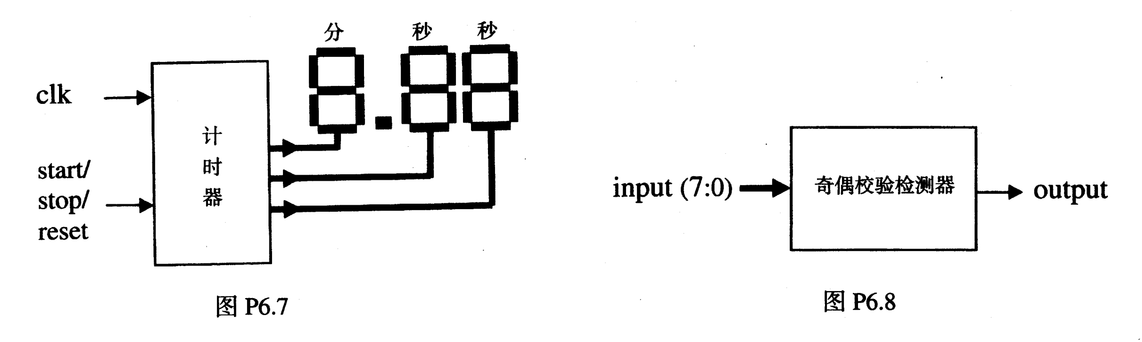


图 P6.7

图 P6.8

表 P6.9

din(7: 1)中'1'的个数	count(2: 0)
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

6.10 **密度编码器**。设计一个编码器，其输入矢量（din）有 7 位；当输入矢量中出现 n（0≤n≤7）个'1'时，那么输出矢量 dout(n) = 1，dout(m) = 0，m≠n。表 P6.10 列出了所有可能的情况。

表 P6.10

din(7: 1)中'1'的个数	dout(7: 0)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

6.11 **多路复用器**。试用顺序代码来实现习题 5.1 中的多路复用器。如果可能，试给出多种实现方法。

6.12 **矢量移位寄存器**。试用顺序代码来实现例 5.6 中的矢量移位寄存器。如果可能，试给出多种实现方法。

6.13 **ALU**。试用顺序代码来实现例 5.5 中的 ALU。如果可能，试给出多种实现方法。

6.14 有符号数（无符号数）加法器/减法器。试用顺序代码来实现习题 5.5 中的有符号数（无符号数）加法器/减法器，并使得这个电路具有通用性。

6.15 比较器。试用顺序代码实现习题 5.8 中的比较器。

6.16 逐级进位加法器。分析例 6.8 中的逐级进位加法器，回答下面的问题：

（1）在第二种实现方案中，为什么不能用一个简单的语句 temp := c0 替代第 17 行~第 19 行的 IF 语句。

（2）例 6.8 中的电路是纯组合逻辑的。试使用并发代码来实现这个电路，并分析其仿真结果。

6.17 D 触发器。对于图 6.1 给出的带异步复位端的触发器，下面给出了许多实现方案，试确定它们是否可以按要求实现电路，并简要解释。

```
---------------------------------------------------
LIBRARY  ieee;
USE  ieee.std_logic_1164.all;
---------------------------------------------------
ENTITY dff IS
   PORT (d, clk, rst: IN  BIT;
          q: OUT BIT);
END  dff;
---------------------方案 1----------------------
ARCHITECTURE arch1 OF dff IS
BEGIN
   PROCESS (clk, rst)
   BEGIN
      IF (rst = '1') THEN
         q <= '0';
      ELSIF (clk'EVENT AND clk = '1') THEN
         q <= d;
      END IF;
   END PROCESS;
END arch1;
--------------------方案 2-----------------------
ARCHITECTURE arch2 OF dff IS
BEGIN
   PROCESS (clk)
   BEGIN
      IF (rst = '1') THEN
         q <= '0';
      ELSIF (clk'EVENT AND clk = '1') THEN
         q <= d;
```

```
        END IF;
    END PROCESS;
END arch2;
--------------------方案3-----------------------
ARCHITECTURE arch3 OF dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF (rst = '1') THEN
            q <= '0';
        ELSIF (clk'EVENT) THEN
            q <= d;
        END IF;
    END PROCESS;
END arch3;
--------------------方案4-----------------------
ARCHITECTURE arch4 OF dff IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF (rst = '1') THEN
            q <= '0';
        ELSIF (clk = '1') THEN
            q <= d;
        END IF;
    END PROCESS;
END arch4;
--------------------方案5-----------------------
ARCHITECTURE arch5 OF dff IS
BEGIN
    PROCESS (clk, rst, d)
    BEGIN
        IF (rst = '1') THEN
            q <= '0';
        ELSIF (clk = '1') THEN
            q <= d;
        END IF;
    END PROCESS;
END arch5;
-------------------------------------------------
```

第 7 章　信号和变量

VHDL 提供了 SIGNAL 和 VARIABLE 这两种对象来处理非静态数据，同时提供了 CONSTANT 和 GENERIC 这两种对象来处理静态数据，其中 GENERIC 在第 4 章中已经介绍过，本章将介绍 SIGNAL（信号）、VARIABLE（变量）和 CONSTANT（常量）。

常量和信号是全局的，既可以用在顺序执行的代码中，也可以用在并发执行的代码中。相比较而言，变量是局部的，只能用在顺序代码中（包括 PROCESS，FUNCTION 或 PROCEDURE 的内部），并且它们的值是不能直接向外传递的。

在某些情况下，很难决定选择信号还是变量，所以本章专门用一节来阐述这个问题。本章还将讨论信号和变量的赋值问题，并分析经过编译器编译后所产生的寄存器的数量。

7.1　常量

常量用来确定默认值，其格式如下所示：

```
CONSTANT 常量名: type :=  值;
```

例

```
CONSTANT set_bit : BIT := '1';
CONSTANT datamemory : memory := (('0','0','0','0'),
                                  ('0','0','0','1'),
                                  ('0','0','1','1'));
```

常量可以在包集、实体或结构体中声明。在包集中声明的常量是真正全局的，可以被所有调用该包集的实体使用。定义在实体中的常量对于该实体的所有结构体而言是全局的。类似地，定义在结构体中的常量仅仅在该结构体内部是全局的。

7.2　信号

VHDL 中的信号代表的是逻辑电路中的“硬”连线，既可以用于电路单元的输入/输出端口，也可以用于电路内部各单元之间的连接。实体的所有端口都默认为信号。信号定义的一般格式如下：

```
SIGNAL name: type [range] [ :=  initial_value];
```

例

```
SIGNAL control: BIT := '0';
SIGNAL count: INTEGER RANGE 0 TO 100;
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

有关信号的最重要的一点是，当信号用在顺序描述语句（如 PROCESS 内部）中时，其值不是立刻更新的，信号值是在相应的进程、函数或过程完成之后才进行更新的。

对于信号，我们采用的赋值符号是 <= 。在上面进行信号定义的语法结构中，对信号赋初始值的操作是不可综合的，只能用来进行仿真。

这里可能出现的另一个问题是对同一个信号进行多重赋值。遇到这种情况，编译器可能会给出错误警告并退出综合过程，或者仅认为最后一次赋值是有效的，从而编译产生一个错误的电路。下面的例题对信号和变量的使用进行了分析。

例 7.1 “1”计数器#1

我们设计了一个电路来计算在一个二进制矢量中'1'的个数（与习题 6.9 相同）。在后面给出的代码中，对同一个信号（temp）在第 15 行和第 18 行都进行了赋值。由于信号的赋值不是立刻生效的，第 15 行赋予信号的值只有在该进程结束后才会真正更新，因此当代码执行到第 18 行时，temp 的初值可能是 temp 所有可能取值中的任何一个，这必将导致发生错误。在这种情况下，推荐用变量来记录中间值，因为变量的赋值是立刻生效的（参见例 7.2）。

仔细分析本例题中的代码可以发现，不使用内部信号 temp 也可以完成设计（第 11 行），因为 ones 本身就是一个端口，可以被直接使用。但如果要这样做，必须将 ones 的端口方向由"OUT"变为"BUFFER"（第 7 行），因为 ones 需要被赋值并且在内部被调用。

```
1  ---------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY count_ones IS
6     PORT (din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            ones: OUT INTEGER RANGE 0 TO 8);
8  END count_ones;
9  ---------------------------------------------
10 ARCHITECTURE not_ok OF count_ones IS
11    SIGNAL temp: INTEGER RANGE 0 TO 8;
12    BEGIN
13 PROCESS (din)
14      BEGIN
15         temp <= 0;
```

```
16          FOR i IN 0 TO 7 LOOP
17             IF (din(i) = '1') THEN
18                temp <= temp+1;
19             END IF;
20          END LOOP;
21          ones <= temp;
22       END PROCESS;
23 END not_ok;
24 -------------------------------------------
```

7.3 变量

与信号和常量相比，变量仅用于局部的电路描述。它只能在 PROCESS，FUNCTION 和 PROCEDURE 内部使用，而且对它的赋值是立即生效的，所以新的值可以在下一行代码中立即使用。

变量说明的格式如下：

```
VARIABLE name: type [range][ := 初始值];
```

例

```
VARIABLE control: BIT := '0';
VARIABLE count: INTEGER RANGE 0 TO 100;
VARIABLE y: STD_LOGIC_VECTOR (7 DOWNTO 0) := "10001000";
```

变量赋值使用的符号是 " := "（如 count := 35;）。与信号一样，对信号赋初值的操作是不可综合的，仅能用在仿真中。

例 7.2 "1"计数器#2

让我们再来考虑例 7.1 中的问题。下面所采用的方法与前面方法的惟一不同是用内部变量代替了信号。因为变量值的更新是立刻生效的，它的初始值可以设定，并且编译器允许对变量多次赋值。图 7.1 给出了相应的的仿真结果。

```
1 ---------------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------------
5 ENTITY count_ones IS
6    PORT (din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7          ones: OUT INTEGER RANGE 0 TO 8);
8 END count_ones;
9 ---------------------------------------------
```

```
10 ARCHITECTURE ok OF count_ones IS
11 BEGIN
12    PROCESS (din)
13       VARIABLE temp: INTEGER RANGE 0 TO 8;
14    BEGIN
15       temp := 0;
16       FOR i IN 0 TO 7 LOOP
17          IF (din(i) = '1') THEN
18             temp := temp+1;
19          END IF;
20       END LOOP;
21       ones <= temp;
22    END PROCESS;
23 END ok;
24 -------------------------------------------------
```

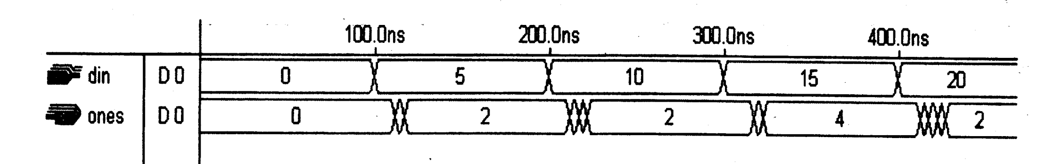


图 7.1 例 7.2 的仿真结果

7.4 信号和变量的比较

综上所述，在变量和信号之间做出选择并不总是很容易的。它们的主要区别总结在表 7.1 中。

表 7.1 信号和变量的比较

	信号	变量
赋值符号	<=	:=
功能	表示电路内部连接	表示局部信息
范围	全局	局部（仅在相应的进程、函数和过程中使用）
行为	在顺序代码中，信号值的更新不是即时的，新的值要在进程、函数或过程完成以后才有效	即时更新（新的值在代码的下一行就生效）
用途	用于包集、实体或结构体中。在实体中，所有端口默认为信号	仅用于顺序描述代码中（进程、函数或过程中）

我们再次强调对变量的赋值是立刻生效的，而信号的赋值则不能立刻生效。一般地，只有当信号所在的 PROCESS 内的操作完成一遍后，信号值的更新才会生效。下面的例子将更进一步说明这一点以及信号与变量的其他区别。

例 7.3 关于多路复用器的对比设计

这里将设计与例 5.2 中相同的多路复用器（如图 7.2 所示），用于对比采用信号实现与采用变量实现的区别。

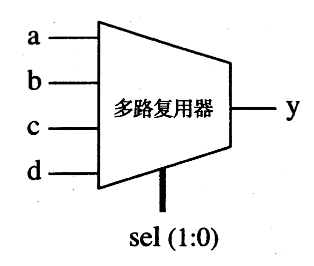


图 7.2 例 7.3 中的多路复用器

```
1  ----------方案 1: using a SIGNAL(not ok)-----------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY mux IS
6     PORT (a, b, c, d, s0, s1: IN STD_LOGIC;
7             y: OUT STD_LOGIC);
8  END  mux;
9  -------------------------------------------------
10 ARCHITECTURE not_ok OF mux IS
11    SIGNAL sel: INTEGER RANGE 0 TO 3;
12 BEGIN
13    PROCESS (a, b, c, d, s0, s1)
14    BEGIN
15       sel <= 0;
16       IF  (s0 = '1') THEN sel <= sel+1;
17       END IF;
18       IF (s1 = '1') THEN sel <= sel+2;
19       END IF;
20       CASE sel IS
21          WHEN 0 =>  y <= a;
22          WHEN 1 =>  y <= b;
23          WHEN 2 =>  y <= c;
```

```
24            WHEN 3 =>  y <= d;
25          END CASE;
26      END PROCESS;
27  END not_ok;
28  -------------------------------------------------
```

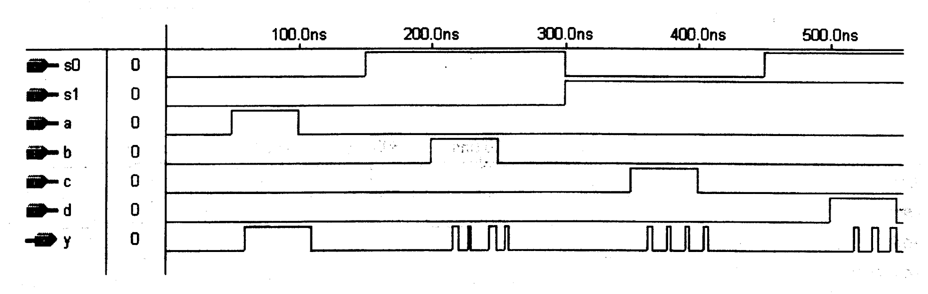
```
1   ----------方案 2: using a VARIABLE (ok)------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   ------------------------------------------------
5   ENTITY mux IS
6      PORT (a, b, c, d, s0, s1: IN STD_LOGIC;
7              y: OUT STD_LOGIC);
8   END  mux;
9   ------------------------------------------------
10  ARCHITECTURE ok OF mux IS
11  BEGIN
12     PROCESS (a, b, c, d, s0, s1)
13        VARIABLE sel: INTEGER RANGE 0 TO 3;
14     BEGIN
15        sel := 0;
16        IF  (s0 = '1') THEN sel := sel+1;
17        END IF;
18        IF (s1 = '1') THEN sel := sel+2;
19        END IF;
20        CASE sel IS
21           WHEN 0 =>  y <= a;
22           WHEN 1 =>  y <= b;
23           WHEN 2 =>  y <= c;
24           WHEN 3 =>  y <= d;
25        END CASE;
26     END PROCESS;
27  END ok;
28  ------------------------------------------------
```

在使用信号时普遍出现的问题是忽略了信号值的更新不能立刻生效这一特点。因此，在第一种实现方案中，虽然在第 15 行对 sel 赋值为'0'，但它不能立刻生效，此时 sel 的值可能是 0~3 中的任何一个，所以此后的第 16 行的赋值操作 sel <= sel+1 的结果也是不确定的。如果使用变量就不会出现这个问题，因为它的更新是立刻生效的。

方案 1 中存在的另一个问题是有些综合工具不能支持对同一个信号的多次赋值。一般来说，

进程中只允许对一个信号进行一次赋值操作，所以有些软件工具只考虑最后一次赋值，有些工具只是给出错误信息，然后结束编译。

两种设计方案的仿真结果如图 7.3 所示，图中上半部分是错误设计的仿真结果，下半部分是正确设计的仿真结果。



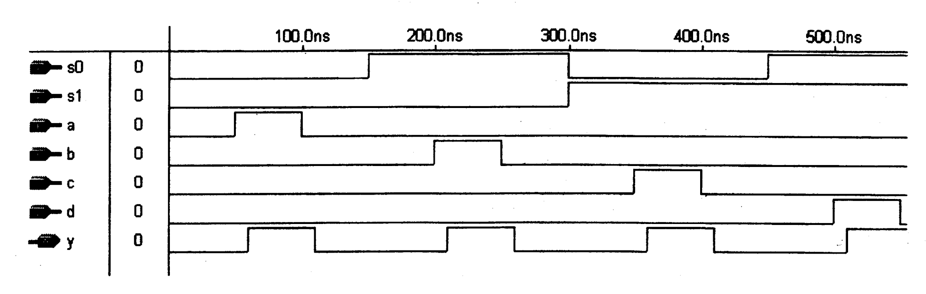


图 7.3　例 7.3 的仿真结果

例 7.4　带 q 和 qbar 的 DFF #1

下面将对图 7.4 给出的 DFF 进行分析。它与例 6.1 的不同之处在于具有复位端（reset）和反相输出端（qbar）。端口 qbar 的引入有助于对信号赋值进行更进一步的理解（注意，在默认情况下端口都是信号类型的）。

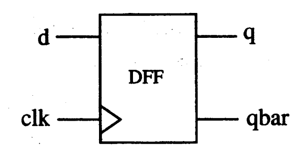
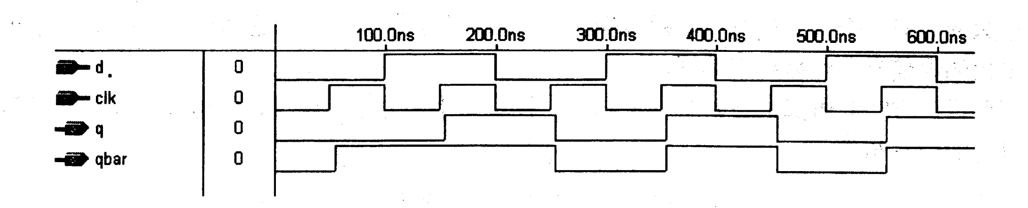


图 7.4　例 7.4 中的 D 触发器

在方案 1 中，q <= d（第 16 行）和 qbar <= NOT q（第 17 行）都出现在 PROCESS 中，所以新赋给它们的值只有在进程结束后才生效。这时 qbar 就会出现问题，因为 q 的值还没有更新，所以 qbar 是将以前的 q 值取反后输出的。换句话说，正确的 qbar 值输出会延时一个时钟周期，这显然是与设计的初衷相违背的。它的仿真波形如图 7.5 上半部分所示。

在方案 2 中，我们将 qbar <= NOT q（第 30 行）放到进程以外，它与进程是并发的，当 q 发生变化后，qbar 的值立刻可以实现更新。它的电路仿真波形在图 7.5 的下半部分给出。

```
1   ----------------方案 1: not_ok-----------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
```



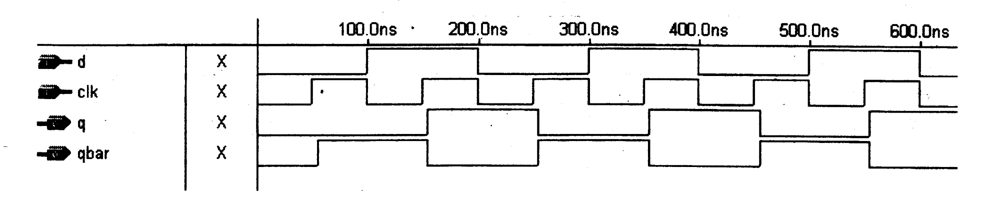


图 7.5　例 7.4 的仿真结果

```
4  --------------------------------------------------
5  ENTITY dff IS
6     PORT (d, clk: IN STD_LOGIC;
7           q: BUFFER STD_LOGIC;
8           qbar: OUT STD_LOGIC);
9  END dff;
10 --------------------------------------------------
11 ARCHITECTURE not_ok OF dff IS
12 BEGIN
13    PROCESS (clk)
14    BEGIN
15       IF (clk'EVENT AND clk = '1') THEN
16          q <= d;
17          qbar <= NOT q;
18       END IF;
19    END PROCESS;
20 END not_ok;
21 --------------------------------------------------
```

```
1  ------------------方案 2: ok----------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY dff IS
6     PORT (d, clk: IN STD_LOGIC;
```

```
7          q: BUFFER STD_LOGIC;
8          qbar: OUT STD_LOGIC);
9  END dff;
10 ---------------------------------------------------
11 ARCHITECTURE ok OF dff IS
12 BEGIN
13    PROCESS (clk)
14    BEGIN
15       IF (clk'EVENT AND clk = '1') THEN
16          q <= d;
17       END IF;
18    END PROCESS;
19    qbar <= NOT q;
20 END ok;
21 ---------------------------------------------------
```

例 7.5　分频器

在这个例子中将设计对时钟进行 6 分频的电路（见图 7.6）。我们特意设计了两个输出，一个基于信号（count1），另一个基于变量（count2）。为了使两者都工作正常（具有如图 7.7 所示的仿真结果），要求填写代码中的两处空白，并做出解释。

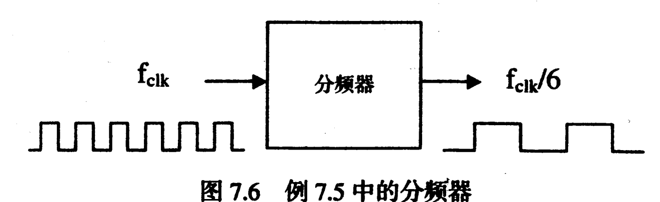


图 7.6　例 7.5 中的分频器

```
1  ---------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  ENTITY freq_divider IS
6     PORT (clk: IN STD_LOGIC;
7           out1, out2: BUFFER STD_LOGIC);
8  EDN freq_divider;
9  ---------------------------------------------------
10 ARCHITECTURE example OF freq_divider IS
11    SIGNAL count1: INTEGER RANGE 0 TO 7;
12 BEGIN
13    PROCESS (clk)
```

```
14        VARIABLE count2: INTEGER RANGE 0 TO 7;
15     BEGIN
16        IF (clk'EVENT AND clk = '1') THEN
17           count1 <= count1+1;
18           count2 := count2+1;
19           IF (count1 =?) THEN
20              out1 <= NOT out1;
21              count1 <= 0;
22           END IF;
23           IF (count2=?) THEN
24              out2 <= NOT out2;
25              count2 := 0;
26           END IF
27        END IF;
28     END PROCESS;
29 END example;
30 --------------------------------------------------
```

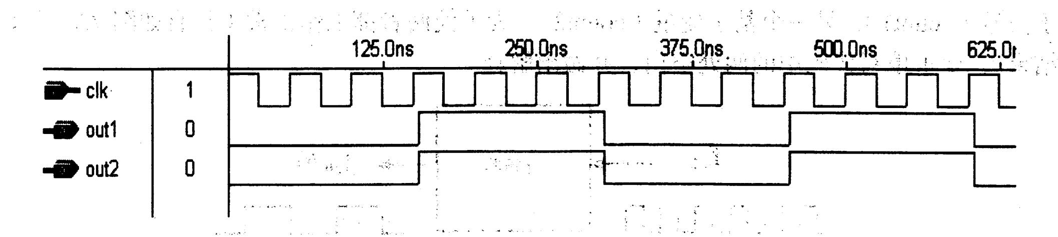


图7.7　例7.5的仿真结果

7.5　寄存器的数量

这一节将讨论编译器对不同风格的代码进行编译时产生寄存器的数量，其目的不仅是为了解使用什么方法可以减少寄存器的数量，更是为了知道代码是否可以实现预期的结果。

当一个信号的赋值是以另一个信号的跳变为条件时，或者说当发生同步赋值时，该信号经过编译后就会生成寄存器。这样的同步赋值只能在进程、函数或过程中出现（一般跟在 IF signal' EVENT…或 WAIT UNTIL...等语句之后）。

如果变量的值没有被进程（函数或过程）以外的代码调用，那么不一定会生成寄存器。如果一个变量是在一个信号跳变时被赋值的，并且该值最终又被赋给了另外的信号（信号是全局的，可以进行数值传递），那么综合后就会产生寄存器。如果一个变量在还没有进行赋值操作时已被使用，那么也会在综合时产生寄存器。下面的例子将有助于理解这几点。

例　在下面所示的进程中，output1 和 output2 都被寄存了。因为两者都是在另一个信号（clk）跳变时被赋值的。

```
PROCESS (clk)
BEGIN
   IF (clk'EVENT AND clk = '1') THEN
      output1 <= temp;       -- output1 被存储
      output2 <= a;          -- output2 被存储
   END IF;
END PROCESS;
```

例　在下面的 PROCESS 中，只有 output1 是寄存器输出的，output2 是组合逻辑输出的。

```
PROCESS (clk)
BEGIN
   IF (clk'EVENT AND clk = '1') THEN
      output1 <= temp;       -- output1 被存储
   END IF;
   output2 <= a;             -- output2 未被存储
END PROCESS;
```

例　在下面的 PROCESS 中，temp（变量）会造成信号 x 被存储。

```
PROCESS (clk)
   VARIABLE temp: BIT;
BEGIN
   IF (clk'EVENT AND clk = '1') THEN
      temp := a;
   END IF;
   x <= temp;                -- temp 促使 x 被存储
END PROCESS;
```

其他例子将在后面给出，目的是为了说明在什么时候以及为什么信号和变量赋值时会生成寄存器。

例 7.6　带 q 和 qbar 的 DFF #2

现在再次对图 7.4 给出的 DFF 进行分析。下面给出的两段代码的功能都是正确的，其区别是两种实现方案综合后所生成触发器的数量不一样。方案 1 有两个同步的信号赋值（第 16 行和第 17 行），所以有两个触发器。方案 2 并不是这样的，其中一个赋值（第 19 行）不是同步的。两种实现方案对应的电路图见图 7.8。

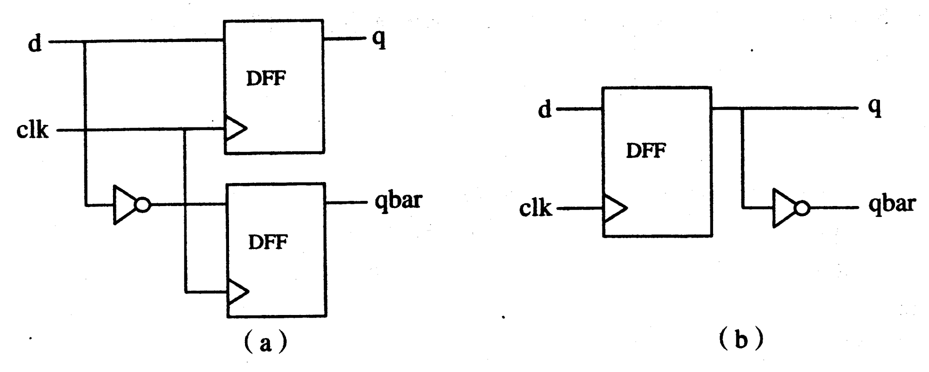


图 7.8 例 7.6 中代码对应的电路。(a) 与第一段代码对应的电路；(b) 与第二段代码对应的电路

```
1  ---------------方案 1: Two DFFs--------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY dff IS
6     PORT (d, clk: IN STD_LOGIC);
7           q: BUFFER STD_LOGIC;
8           qbar: OUT STD_LOGIC);
9  END dff;
10 --------------------------------------------------
11 ARCHITECTURE two_dff OF dff IS
12 BEGIN
13    PROCESS (clk)
14    BEGIN
15       IF (clk'ENENT AND clk = '1') THEN
16          q <= d;          --generates a register
17          qbar <= NOT d;   --generates a register
18       END IF;
19    END PROCESS;
20 END two_dff;
21 --------------------------------------------------
```

```
1  ---------------方案 2: One DFF--------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  --------------------------------------------------
```

```
5   ENTITY dff IS
6      PORT (d, clk: IN STD_LOGIC);
7            q: BUFFER STD_LOGIC;
8            qbar: OUT STD_LOGIC);
9   END dff;
10  --------------------------------------------------
11  ARCHITECTURE one_dff OF dff IS
12  BEGIN
13     PROCESS (clk)
14     BEGIN
15        IF (clk'ENENT AND clk = '1') THEN
16           q <= d;      --generates a register
17        END IF;
18     END PROCESS;
19     qbar <= NOT q;     --uses logic gate (no register)
20  END one_dff;
21  --------------------------------------------------
```

例 7.6 指出了一个非常重要的问题：如果不对代码进行有效合理的组织，那么代码编译后会生成额外的寄存器。在实际应用中有一种很有趣的情况，对于某些类型的 CPLD/FPGA 器件，当 q 和 qbar 直接连接到芯片的引脚上时，CPLD/FPGA 的布局布线器在对综合后的网表进行优化的过程中会生成两个额外的寄存器，而综合工具生成的报告文件中显示仍然只有一个寄存器。在习题 7.7 中将对此做进一步的讨论。

例 7.7　计数器

在本例题中将分析如图 7.9 所示的模 8 计数器。下面给出了两种实现方案。方案 1 中使用了对变量进行同步赋值的方案（方案 1 的第 14 行~第 15 行），方案 2 中使用了对信号进行同步赋值（方案 2 的第 13 行~第 14 行）的方案。

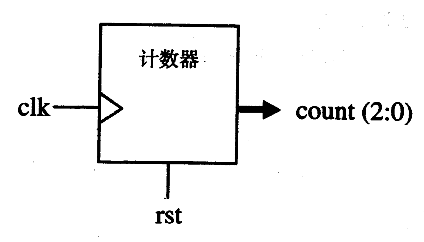


图 7.9　例 7.7 中的模 8 计数器

无论哪一种方案，都需要使用寄存器来保存 3 位的当前计数值。方案 1 中使用了变量（temp），但同样会综合生成寄存器，因为 temp 是在时钟上升沿到达时赋值的，而且它的值又赋给了信号 count，这与前面所讲的内容是一致的。

在方案 2 中仅仅使用了信号。注意，由于没有使用辅助信号，count 既是输出端口，又要用于内部操作，所以必须说明为 BUFFER 模式的端口。

另外需要指出的是，无论在方案 1 还是方案 2 中，都没有进行 std_logic_1164 包集的声明，根本原因是本例题中没有用到 std_logic 数据类型。

```
1  ------------方案 1: with a VARIABLE ---------------
2  ENTITY counter IS
3     PORT (clk, rst: IN BIT;
4           count: OUT INTEGER RANGE 0 TO 7);
5  END counter;
6  --------------------------------------------------
7  ARCHITECTURE counter OF counter IS
8  BEGIN
9     PROCESS (clk, rst)
10       VARIABLE temp: INTEGER RANGE 0 TO 7;
11    BEGIN
12       IF (rst = '1') THEN
13          temp := 0;
14       ELSIF (clk'EVENT AND clk = '1') THEN
15          temp := temp+1;
16       END IF;
17       count <= temp;
18    END PROCESS;
19 END counter;
20 --------------------------------------------------
```

```
1  ----------方案 2: with SIGNALS only -------
2  ENTITY counter IS
3     PORT (clk, rst: IN BIT;
4           count: BUFFER INTEGER RANGE 0 TO 7);
5  END counter;
6  -----------------------------------------------
7  ARCHITECTURE counter OF counter IS
8  BEGIN
9     PROCESS (clk, rst)
10    BEGIN
11       IF (rst = '1') THEN
12          count <= 0;
13       ELSIF (clk'EVENT AND clk = '1') THEN
```

```
14            count <= count+1;
15         END IF;
16      END PROCESS;
17 END counter;
18 --------------------------------------------------
```

上面两种方案的仿真结果如图 7.10 所示。

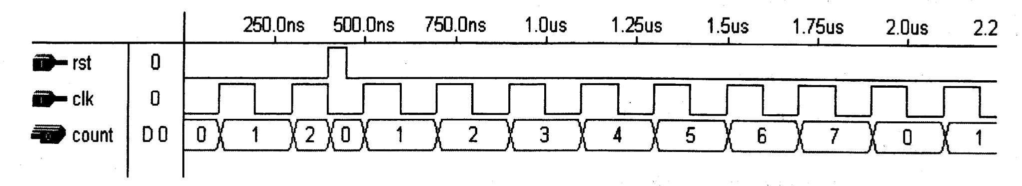


图 7.10　例 7.7 的仿真结果

例 7.8　移位寄存器#1

现在分析分别使用信号和变量来设计如图 7.11 所示的 4 位移位寄存器时需要考虑的问题。当然，如果方法正确，两者应该有相同的结果。

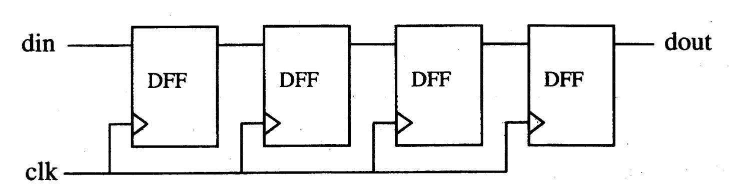


图 7.11　例 7.8 中的移位寄存器

在方案 1 中，使用了 3 个变量（第 10 行的 a，b 和 c）。在第 13 行~第 16 行中，信号 dout，变量 c，变量 b 和变量 a 被依次赋值，由于 c 的值被赋给 dout 之前没有对 c 赋过值，这样的代码经过编译后就会生成寄存器，用于存储 PROCESS 上一次执行后各个变量的值。

在方案 2 中，变量被信号所代替，信号是依次顺序赋值的（第 13 行~第 16 行）。整个电路是在时钟上升沿的触发下进行工作的，编译后将会生成 4 个寄存器。

最后，在方案 3 中，使用了与方案 1 中同样的变量，但赋值顺序是相反的（第 13 行~第 16 行）。由于对变量的赋值是立刻生效的，所以第 13 行~第 15 行可以简化成一行，等效于 c := din。c 的值在下一行（第 16 行）赋给了信号 dout，因而最终会生成一个寄存器。显然这不是我们所期望的结果。

方案 1 和方案 2 的仿真结果显示在图 7.12 的上半部分，可以看出 dout 与 din 的波形完全相同，只是延迟了 4 个时钟周期。图 7.12 的下半部分是方案 3 的仿真结果，可以看出 dout 与 din 的波形完全相同，但只延迟了 1 个时钟周期。

```
1  -------------------方案1: ------------------------
2  ENTITY shift IS
3     PORT (din, clk: IN BIT;
4           dout: OUT BIT);
5  END shift;
6  --------------------------------------------------
7  ARCHITECTURE shift OF shift IS
8  BEGIN
9     PROCESS (clk)
10       VARIABLE a, b, c: BIT;
11    BEGIN
12       IF (clk'EVENT AND clk = '1') THEN
13          dout <= c;
14          c := b;
15          b := a;
16          a := din;
17       END IF;
18    END PROCESS;
19 END shift;
20 --------------------------------------------------
1  --------------------方案2: ----------------------
2  ENTITY shift IS
3     PORT (din, clk: IN BIT;
4           dout: OUT BIT);
5  END shift;
6  --------------------------------------------------
7  ARCHITECTURE shift OF shift IS
8     SIGNAL a, b, c: BIT;
9  BEGIN
10    PROCESS (clk)
11    BEGIN
12       IF (clk'EVENT AND clk = '1') THEN
13          a <= din;
14          b <= a;
15          c <= b;
16          dout <= c;
17       END IF;
18    END PROCESS;
19 END shift;
```

```
20 -------------------------------------------------------
1  -------------------方案 3: -----------------------
2  ENTITY shift IS
3     PORT (din, clk: IN BIT;
4           dout: OUT BIT);
5  END shift;
6  -------------------------------------------------------
7  ARCHITECTURE shift OF shift IS
8  BEGIN
9     PROCESS (clk)
10       VARIABLE a, b, c: BIT;
11    BEGIN
12       IF (clk'EVENT AND clk = '1') THEN
13          a := din;
14          b := a;
15          c := b;
16          dout <= c;
17       END IF;
18    END PROCESS;
19 END shift;
20 -------------------------------------------------------
```


图 7.12　例 7.8 的仿真结果

例 7.9　移位寄存器 #2

在本例题中将介绍设计移位寄存器的常规方法。图 7.13 显示的移位寄存器除增加了一个复位输入端口以外与例 7.8 中的完全相同。与先前一样，输出位（q）将比输入位（d）延迟 4 个时钟周期。这里采用的是异步复位方式，当 rst 有效时，所有的触发器都被强制置'0'。

下面给出了两种实现方案，一种是用信号来生成寄存器，另一种是用变量来生成寄存器。两种方案综合出来的电路是相同的。在方案 1 中，由于对信号的赋值是在时钟上升沿的触发下进行的，所以综合后会生成寄存器。在方案 2 中，变量的赋值也是靠时钟的跳变来触发的，由于信号的值最终赋予了输出端口信号 q，所以也会产生所期望的寄存器。

上面两种方法的仿真结果如图 7.14 所示，显然它们是符合设计要求的。

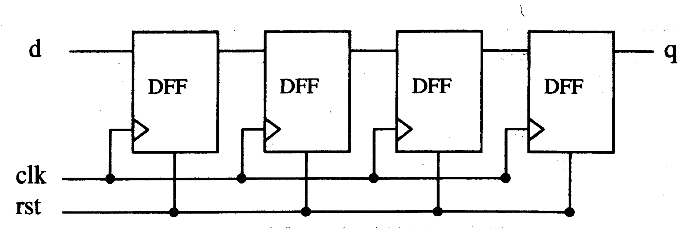


图 7.13　例 7.9 中的移位寄存器

现在可以回顾一下第 6 章例题中给出的信号和变量的用法。第 8 章将通过一系列例子来体会正确理解信号与变量之间的区别的重要性，如果不能正确地使用信号和变量，就会生成错误的电路。

```
1  ------方案 1: with an internal SIGNAL-------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY shiftreg IS
6     PORT (d, clk, rst: IN STD_LOGIC;
7           q: OUT STD_LOGIC);
8  END shiftreg;
9  -------------------------------------------------
10 ARCHITECTURE behavior OF shiftreg IS
11    SIGNAL internal: STD_LOGIC_VECTOR(3 DOWNTO 0);
12 BEGIN
13    PROCESS (clk, rst)
14    BEGIN
15       IF (rst = '1') THEN
16          internal <= (OTHERS => '0');
17       ELSIF (clk'EVENT AND clk = '1') THEN
18          internal <= d & internal(3 DOWNTO 1);
19       END IF;
20    END PROCESS;
21    q <= internal(0);
22 END behavior;
23 -------------------------------------------------
1  -----方案 2: with an internal VARIABLE-0----------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY shiftreg IS
```

```
6      PORT (d, clk, rst: IN STD_LOGIC;
7             q: OUT STD_LOGIC);
8   END shiftreg;
9   ------------------------------------------------
10  ARCHITECTURE behavior OF shiftreg IS
11  BEGIN
12     PROCESS (clk, rst)
13        VARIABLE internal: STD_LOGIC_VECTOR(3 DOWNTO 0);
14     BEGIN
15        IF (rst = '1') THEN
16           internal := (OTHERS => '0');
17        ELSIF (clk'EVENT AND clk = '1') THEN
18           internal := d & internal(3 DOWNTO 1);
19        END IF;
20        q <= internal(0);
21     END PROCESS;
22  END behavior;
23  ------------------------------------------------
```

图 7.14　例 7.9 的仿真结果

7.6　习题

7.1　根据给出的 VHDL 对象判断赋值方式是否正确。下列给出了一些 VHDL 对象：

```
CONSTANT max: INTEGER := 10;
SIGNAL x: INTEGER RANGE -10 TO 10;
SIGNAL y: BIT_VECTOR(15 DOWNTO 0);
VARIABLE z: BIT;
```

判断下面的赋值中哪些是合法的（建议回顾第 3 章的相关内容）：

```
x <= 5;
x <= y(5);
```

```
z <= '1';
z := y(5);
WHILE i IN 0 TO max LOOP ...
FOR i IN 0 TO x LOOP...
G1: FOR i IN 0 TO max GENERATE...
G1: FOR i IN 0 TO x GENERATE...
```

7.2　数据时延。图 P7.2 给出了可编程数据时延电路的原理图。输入（d）和输出（q）都是 4 位总线。q 是将 d 进行延迟后输出的结果，延迟的时钟周期数由 sel 来确定。

（a）写出该电路的 VHDL 代码；

（b）你认为该设计方法会生成多少个寄存器？

（c）综合你设计的代码并分析报告文件，验证实际使用的寄存器数量与预计的是否一致。

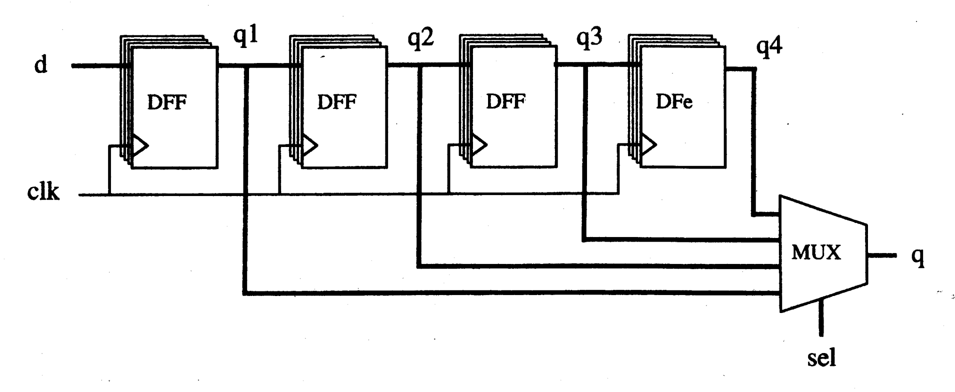


图 P7.2

7.3　带 q 和 qbar 的 DFF #1。我们希望实现与例 7.4（见图 7.4）相同的触发器。这里，在代码中引入了辅助信号 temp。分析下面的 3 种实现方案，并判断在哪种情况下 q 和 qbar 能够正常工作并进行简要说明。

```
--------------------------------------------------
ENTITY dff IS
   PORT (d, clk: IN BIT;
         q, qbar: BUTTER BIT);
END dff;
----------------------方案 1----------------------
ARCHITECTURE arch1 OF dff IS
   SIGNAL temp: BIT;
BEGIN
   PROCESS (clk)
   BEGIN
```

```
        IF (clk'EVENT AND clk = '1') THEN
            temp <= d;
            q <= temp;
            qbar <= NOT temp;
        END IF;
    END PROCESS;
END arch1;
---------------------方案 2---------------------
ARCHITECTURE arch2 OF dff IS
    SIGNAL temp: BIT;
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'EVENT AND clk = '1') THEN
            temp <= d;
        END IF;
        q <= temp;
        qbar <= NOT temp;
    END PROCESS;
END arch2;
---------------------方案 3---------------------
ARCHITECTURE arch3 OF dff IS
    SIGNAL temp: BIT;
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'EVENT AND clk = '1') THEN
            temp <= d;
        END IF;
    END PROCESS;
    q <= temp;
    qbar <= NOT temp;
END arch3;
-----------------------------------------------
```

7.4 **带 q 和 qbar 的 DFF #2**。这里分析的是与习题 7.3 相同的问题，不同的是采用了辅助变量来代替辅助信号。分析下面的 3 种实现方案，并判断在哪种情况下 q 和 qbar 能够正常工作并进行简要说明。

```
---------------------------------------------------
ENTITY dff IS
   PORT (d, clk: IN BIT;
         q: BUGGER BIT;
         qbar: OUT BIT);
END dff;
----------------------方案 1----------------------
ARCHITECTURE arch1 OF dff IS
BEGIN
   PROCESS (clk)
      VARIABLE temp: BIT;
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         temp := d;
         q <= temp;
         qbar <= NOT temp;
      END IF;
   END PROCESS;
END arch1;
----------------------方案 2----------------------
ARCHITECTURE arch2 OF dff IS
BEGIN
   PROCESS (clk)
      VARIABLE temp: BIT;
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         temp := d;
         q <= temp;
         qbar <= NOT q;
      END IF;
   END PROCESS;
END arch2;
----------------------方案 3----------------------
ARCHITECTURE arch3 OF dff IS
BEGIN
   PROCESS (clk)
      VARIABLE temp: BIT;
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
```

```
            temp := d;
            q <= temp;
        END IF;
    END PROCESS;
    qbar <= NOT q;
END arch3;
---------------------------------------------
```

7.5 计数器。参见例 6.2 所示的 4 位计数器，在本习题中，计数器要从"0000"计数到"1111"。

（a）编写 VHDL 代码，然后进行综合和仿真，并验证其是否可以正确工作。

（b）打开综合工具生成的报告文件，确认有 4 个寄存器产生。

（c）根据报告文件，分析所生成的电路是否和图 P7.5 给出的电路一致。给出图 P7.5 中最右侧寄存器输入信号的表达式。

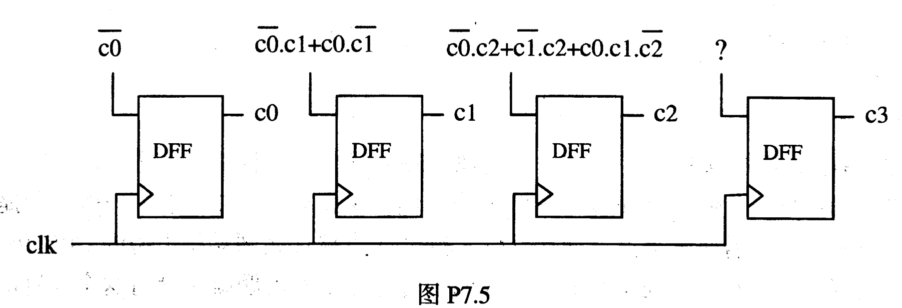


图 P7.5

7.6 通用 m-n 译码器。下面分析例 4.1 中出现的通用 m-n 译码器（见图 P7.6）。代码如下所示，虽然很简捷，但在代码 x <= (sel => '0', OTHERS => '1')中存在缺陷，原因是 sel 不是局部静态信号（它出现在进程的敏感信号列表中）。试修正代码中的错误。

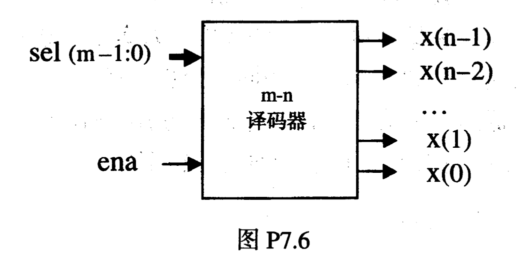


图 P7.6

```
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY decoder IS
```

```
   PORT (ena: IN STD_LOGIC;
         sel: IN INTEGER RANGE 0 TO 7;
         x: OUT STD_LOGIC_VECTOR(0 DOWNTO 0));
END decoder;
-------------------------------------------------
ARCHITECTURE not_ok OF decoder IS
BEGIN
   PROCESS (ena, sel)
   BEGIN
      IF (ena = '0') THEN
         x <= (OTHERS => '1');
      ELSE
         x <= (sel => '0', OTHERS => '1');
      END IF;
   END PROCESS;
END not_ok;
-------------------------------------------------
```

7.7 带 q 和 qbar 的 DFF #3。分析例 7.6 中实现 DFF 的方案 2。我们感兴趣的是在实现过程中所需的寄存器数量。我们已经知道答案是 1 个。然而正如在对例 7.6 进行分析时所提到的，即使综合后的结果是需要 1 个寄存器，但在 q 和 qbar 直接连接到某种类型的 CPLD/FPGA 芯片的输出引脚上时，布局布线工具通常添加两个寄存器。本习题就是针对这个问题的。

（a）用 Quartus II 3.0（见附录 D）来编译例 7.6 中的代码（方案 2）。选择 MAX3000A 或 Cyclone 系列的器件来实现本设计。根据综合报告，确认生成寄存器的数量和综合器实现的方程式，然后与完成布局布线后所得的报告进行对比。

（b）选择 FLEX10K 系列可编程器件，重复上面的过程。

（c）用 ISE 6.1（见附录 B）来编译例 7.6 中的代码（方案 2）。选择 XC9500 或 CoolRunner II 系列器件。在编译完成后进行与上面相同的分析工作。

（d）最后，考虑当寄存器的一个输出不直接与芯片的引脚相连时的情况。如果在下面的代码中引入一个信号 test，重复前面所有的对比分析工作。

```
1  -------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY dff IS
6     PORT (d, clk, test: IN STD_LOGIC;
7           q: BUFFER STD_LOGIC;
8           qbar: OUT STD_LOGIC);
```

```
9  END dff;
10 --------------------------------------------------
11 ARCHITECTURE one_dff OF dff IS
12 BEGIN
13    PROCESS (clk)
14    BEGIN
15       IF (clk'EVENT AND clk = '1') THEN
16          q <= d;
17       END IF;
18    END PROCESS;
19    qbar <= NOT q AND test;
20 END one_dff;
21 --------------------------------------------------
```

第 8 章　状　态　机

有限状态机（FSM）是为时序逻辑电路设计创建的特殊模型技术。这种模型在设计某些类型的系统时非常有用，特别是对那些任务顺序非常明确的电路（如数字控制模块）更是如此。本章首先回顾一下有关 FSM 的基本概念，接下来通过完成一些例题来介绍相应的 VHDL 编码方法。

8.1　引言

图 8.1 是一个状态机示意图，其下半部分是时序逻辑电路，上半部分是组合逻辑电路。

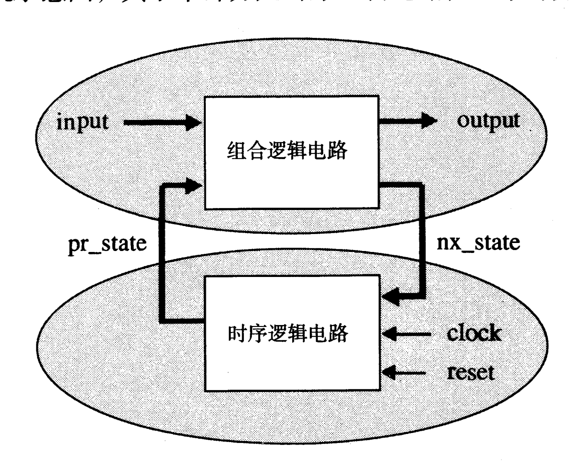


图 8.1　米里（摩尔）型状态机的结构图

组合逻辑电路包含两部分输入：一部分是 pr_state（present state，当前状态），另一部分是实际的外部输入信号。组合逻辑电路送出两部分信号：nx_state（next state，下一个状态）和实际的电路输出信号。

时序逻辑电路包含 3 个输入信号（clock，reset 和 nx_state）以及一个输出信号（pr_state）。因为所有的寄存器都放在这一部分，所以 clock 和 reset 都与这部分电路相连。

如果状态机的输出信号不仅与电路的当前状态有关，还与当前的输入有关，则这种状态机称为米里（Mealy）型状态机。如果状态机的当前输出仅仅由当前状态决定，则称之为摩尔（Moore）型状态机。后面将给出这两种状态机的例子，以进一步说明。

我们将这类电路从结构上划分为两个部分，同样在代码结构上也可以将其划分为两个部分。从 VHDL 代码编写的角度出发，很容易看出：对于时序逻辑电路部分，应该在 PROCESS 内部进

行描述（PROCESS 内部的代码是按照顺序执行的），而对于组合逻辑电路部分，则不需要。根据前面的章节可以知道，顺序代码既可以综合生成时序逻辑，也可以生成组合逻辑，因此如果愿意，上半部分也可以在 PROCESS 中实现。

clock 和 reset 通常应该出现在用来实现时序逻辑电路功能的 PROCESS 的敏感信号列表中（除非不使用 reset 或使用同步 reset，或用 WAIT 来代替 IF）。当 reset 有效时，pr_state 将强制回到系统的初始状态；当 reset 无效时，每当出现时钟的上升沿，寄存器将存储 nx_state，并通过 pr_state 反馈给组合逻辑电路。

虽然在理论上任何时序电路都可以建立状态机模型，但这并不总是一种高效的电路实现方式，如果不加区别地追求建立电路的状态机模型，可能会使代码更加冗长和容易出错。譬如，对于简单的计数器，就不需要使用状态机来实现。如果电路要完成的任务或要实现的功能可以进行完整清晰的排列和归类，那么应该首选使用状态机来实现该电路。设计这类电路时，通常会在 ARCHITECTURE 的开始部分插入一个用户自定义的枚举数据类型，其中包含所有可能出现的系统状态。在分析了一些例题之后，读者会对这种做法更加熟悉。

8.2 设计风格#1

设计者可以使用多种方法来设计 FSM。下面将详细讨论一种结构清晰、易于实现的 FSM 设计风格。使用这种方法进行设计时，FSM 中的时序逻辑部分和组合逻辑部分可以分开独立设计。它需要定义一个枚举数据类型，其内部包含所有 FSM 需要的状态。在介绍完这种设计风格后，将从数据存储的角度对它进行分析，以便于更进一步理解和精简它的结构，这样就引出了设计风格#2。

状态机中时序逻辑部分的设计

参见图 8.1，时序逻辑部分包含寄存器，所以 clock 和 reset 都与之相连。下半部分的输入信号是 nx_state，输出信号是 pr_state。下半部分是时序电路，在 PROCESS 中实现。在 PROCESS 中，任何一种顺序描述语句[IF，WAIT，CASE 或 LOOP（参见第 6 章）]都可以使用。

图 8.1 中下半部分的典型设计模板如下所示：

```
PROCESS (reset, clock)
BEGIN
   IF (reset = '1') THEN
      pr_state <= state0;
   ELSIF (clock'EVENT AND clock = '1') THEN
      pr_state <= nx_state;
   END IF;
END PROCESS;
```

上面的代码非常简单，它包含了决定系统初始状态（state0）的异步复位信号 reset，后面是 nx_state 的同步存储（由时钟上升沿触发），nx_state 将更新当前的 pr_state 值（见图 8.1）。这种 FSM

设计风格的优点是时序电路的设计方法基本上是标准的。

这种设计风格的另一个优点是占用的寄存器数量最少。从前面的 7.5 节可以知道，上面的代码综合得到的寄存器的数量等于对 FSM 所有状态进行编码所需的位数。因此，如果使用默认（二进制）的编码方式（见 8.4 节），则需要的寄存器数量是$\lceil \log_2 n \rceil$，其中 n 是总的状态数，$\lceil\ \rceil$是取大于等于 $\log_2 n$ 的最小整数。

状态机中组合逻辑部分的设计

在图 8.1 中，对于组合逻辑电路而言，通常使用并发代码来实现它的功能，当然也可以在 PROCESS 中使用顺序代码来实现。在下面的设计模板中，仍然使用顺序代码来实现 FSM 的组合逻辑电路部分，其核心是一个 CASE 语句。注意，在这种情况下必须遵守 6.10 节中提出的代码编写规则 1 和规则 2。

```
PROCESS (input, pr_state)
BEGIN
   CASE pr_state IS
      WHEN state0 =>
         IF (input=...) THEN
            output <= <value>;
            nx_state <= state1;
         ELSE...
         END IF;
      WHEN state1 =>
         IF (input=...) THEN
            output <= <value>;
            nx_state <= state2;
         ELSE...
         END IF;
      WHEN state2 =>
         IF (input=...) THEN
            output <= <value>;
            nx_state <= state3;
         ELSE...
         END IF;
      ...
   END CASE;
END PROCESS;
```

可以看到，这段代码也很简单，它只做了两件事情：对输出端口赋值和确定状态机的下一个状态。同样可以看出，它遵循了采用顺序代码设计组合逻辑电路的基本要求（6.10 节中的原则 1

和原则2)，即所有输入信号都必须出现在PROCESS的敏感信号列表中，并且所有输入/输出信号组合都必须完整列出。在整个代码中，由于没有任何信号的赋值是通过其他某个信号的跳变来触发的，所以不会生成寄存器（见7.5节)。

设计风格#1的状态机模板

对于设计风格**#1**，下面给出了完整的设计模板。需要指出的是，除了上面介绍的两个进程之外，模板还包括一个用户自定义的枚举类型，所有可能的状态都列在其中。

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY <entity_name>IS
   PORT (input: IN<data_type>;
         reset, clock: IN STD_LOGIC;
         output: OUT <data_type>);
END <entity_name>;
-------------------------------------------------
ARCHITECTURE <arch_name>OF<entity_name>IS
   TYPE state IS (state0, state1, state2, state3, ...);
   SIGNAL pr_state, nx_state: state;
BEGIN
  -----------------Lower section: ---------------
   PROCESS (reset, clock)
   BEGIN
      IF (reset = '1') THEN
         pr_state <= state0;
      ELSIF (clock'EVENT AND clock = '1') THEN
         pr_state <= nx_state;
      END IF;
   END PROCESS;
   ----------------Upper section: ----------------
   PROCESS (input, pr_state)
   BEGIN
      CASE pr_state IS
      WHEN state0 =>
         IF (input=...) THEN
            output <= <value>;
            nx_state <= state1;
         ELSE...
```

```
         END IF;
       WHEN state1 =>
         IF (input=...) THEN
            output <= <value>;
            nx_state <= state2;
         ELSE...
         END IF;
      WHEN state2 =>
         IF (input=...) THEN
            output <= <value>;
            nx_state <= state3;
         ELSE...
         END IF;
      ...
      END CASE;
    END PROCESS;
END <arch_name>;
```

例 8.1　BCD 计数器

下面是使用摩尔型状态机设计 BCD 计数器的一个实例，它的当前输出仅取决于内部寄存器的当前状态。由于其功能非常简单，因此可以采用前面章节所学习的方法进行设计，当然也可以使用有限状态机来实现。采用有限状态机进行设计时，如果需要的状态很多，枚举所有的状态就变得很不方便。而在常规方法中应用 LOOP 语句就可以避免这一点。

图 8.2 给出了 0~9 环行计数器的状态转移图。这些状态分别为 0，1，…，9，每个状态的名字对应于其输出的二进制数值。

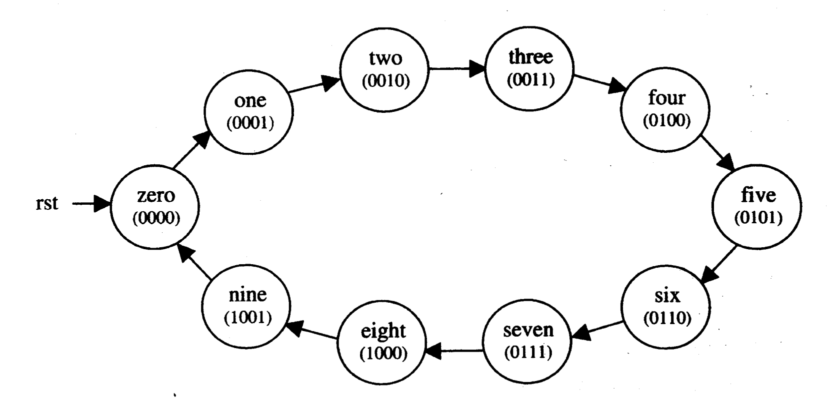


图 8.2　例 8.1 中的状态转移图

下面的 VHDL 代码采用了与设计风格#1 类似的模板。第 11 行~第 12 行是枚举数据类型。状态机的时序逻辑部分出现在第 16 行~第 23 行中，组合逻辑部分出现在第 25 行~第 59 行中。在这个例子中，需要的寄存器个数为$\lceil \log_2 10 \rceil = 4$。

```
1  --------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY counter IS
6     PORT (clk, rst: IN STD_LOGIC;
7            count: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
8  END counter;
9  --------------------------------------------------
10 ARCHITECTURE state_machine OF counter IS
11    TYPE state IS (zero, one, two, three, four,
12       five, six, seven, eight, nine);
13    SIGNAL pr_state, nx_state: state;
14 BEGIN
15    ----------------Lower section: ----------------
16    PROCESS (rst, clk)
17    BEGIN
18       IF (rst = '1') THEN
19           pr_state <= zero;
20       ELSIF (clk'EVENT AND clk = '1') THEN
21           pr_state <= nx_state;
22       END IF;
23    END PROCESS;
24    ----------------Upper section: -----------------
25    PROCESS (pr_state)
26    BEGIN
27       CASE pr_state IS
28          WHEN zero =>
29             count <= "0000";
30             nx_state <= one;
31          WHEN one =>
32             count <= "0001";
33             nx_state <= two;
34          WHEN two =>
35             count <= "0010";
```

```
36              nx_state <= three;
37           WHEN three =>
38              count <= "0011";
39              nx_state <= four;
40           WHEN four =>
41              count <= "0100";
42              nx_state <= five;
43           WHEN five =>
44              count <= "0101";
45              nx_state <= six;
46           WHEN six =>
47              count <= "0110";
48              nx_state <= seven;
49           WHEN seven =>
50              count <= "0111";
51              nx_state <= eight;
52           WHEN eight =>
53              count <= "1000";
54              nx_state <= nine;
55           WHEN nine =>
56              count <= "1001";
57              nx_state <= zero;
58        END CASE;
59     END PROCESS;
60  END state_machine;
61  --------------------------------------------------
```

仿真结果如图 8.3 所示。可以看出，输出从 0 增加到 9，然后再从 0 开始。

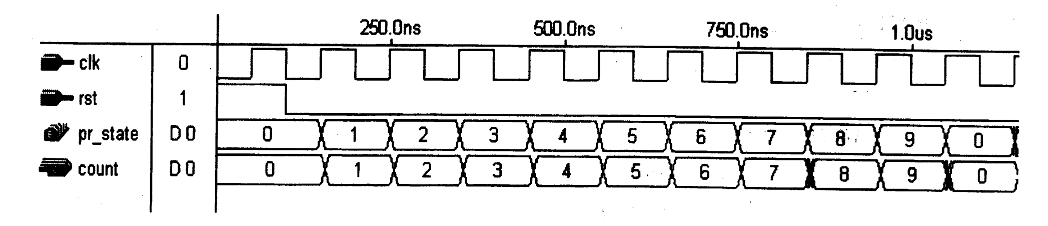


图 8.3　例 8.1 的仿真结果

例 8.2　简单的 FSM #1

图 8.4 给出了一个非常简单的有限状态机。整个设计包括两个状态（状态 A 和状态 B），每次

当发现 d = '1'时，它都会从当前状态跳变到另一个状态。当状态机处于 stateA 时，输出端口 x = a；当状态机处于 stateB 时，有 x = b。

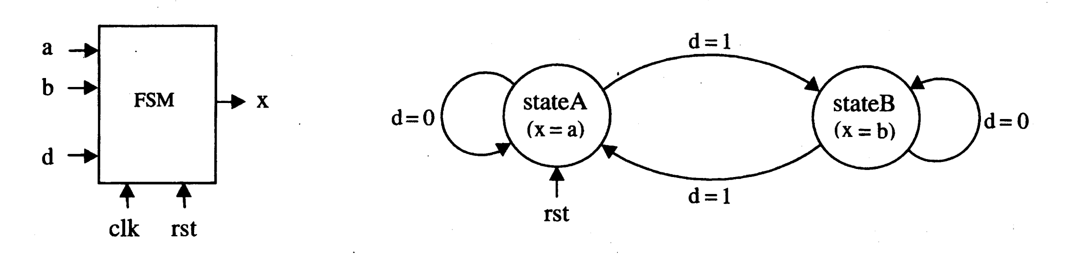


图 8.4　例 8.2 中的状态机

应用设计风格#1 编写的 VHDL 代码如下所示：

```
1  --------------------------------------------------
2  ENTITY simple_fsm IS
3     PORT (a, b, c, d, clk, rst: IN BIT;
4           x: OUT BIT);
5  END simple_fsm;
6  --------------------------------------------------
7  ARCHITECTURE simple_fsm OF simple_fsm IS
8     TYPE state IS(stateA, stateB);
9     SIGNAL pr_state, nx_state: state;
10 BEGIN
11    -------------Lower section: ------------------
12    PROCESS (rst, clk)
13    BEGIN
14       IF (rst = '1') THEN
15          pr_state <= stateA;
16       ELSIF (clk'EVENT AND clk = '1') THEN
17          pr_state <= nx_state;
18       END IF;
19    END PROCESS;
20    -------------Upper section: ------------------
21    PROCESS (a, b, c, d, pr_state)
22    BEGIN
23       CASE pr_state IS
24          WHEN stateA =>
25             x <= a;
26             IF (d = '1') THEN nx_state <= stateB;
```

```
27              ELSE nx_state <= stateA;
28              END IF;
29        WHEN stateB =>
30              x <= b;
31              IF (d = '1') THEN nx_state <= stateA;
32              ELSE nx_state <= stateB;
33              END IF;
34        END CASE;
35     END PROCESS;
36  END simple_fsm;
37  ------------------------------------------------
```

上面代码的仿真结果如图 8.5 所示。可以看出，电路是按照我们所期望的那样在工作的。实际上，通过查看综合过程生成的报告文件，可以发现实现这个电路只需要一个寄存器，因为只需要对两个状态编码。同样值得注意的是组合逻辑部分，输出 x 只取决于当前所处的状态，而与时钟（clk）的跳变无关。如果 x 需要同步输出，那么就需要使用设计风格#2。

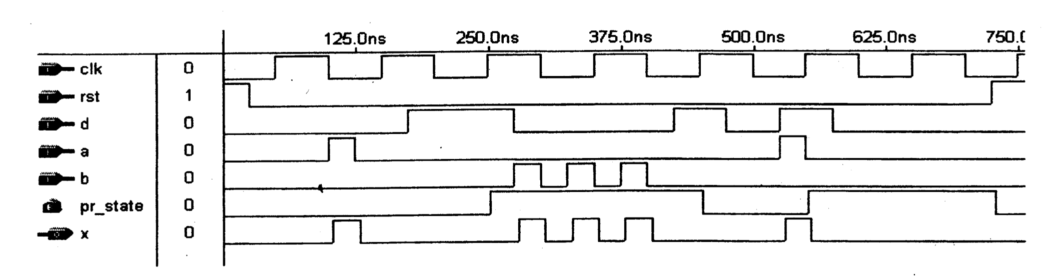


图 8.5　例 8.2 的仿真结果

8.3　设计风格#2

根据前面的分析可以知道，在设计风格#1 中，只存储 pr_state（现态），因此电路具有如图 8.6（a）所示的电路结构。在这样的情况下，如果是米里型状态机（输出只取决于当前输入），那么输出随输入的变化而改变（由组合逻辑决定的异步输出）。

在很多应用中，需要同步的寄存器输出，输出信号只有在时钟边沿出现时才能够更新，此时输出必须先使用寄存器存储起来，如图 8.6（b）所示，这个结构就是设计风格#2。

要实现这个新的结构，只需做一些简单的修改。例如，可以使用辅助信号（如 temp）来计算电路的输出值，但它的值只有在某个时钟边沿出现时才传递给真正的输出信号。这个修改可以在下面的模板中体现出来。

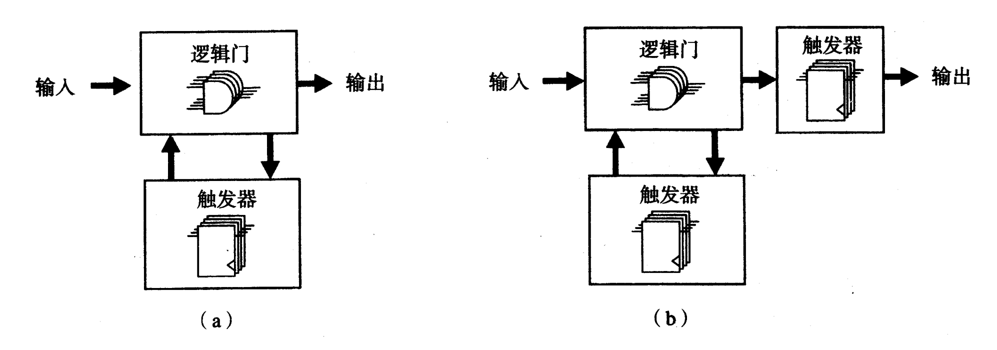


图 8.6 (a)设计风格#1 的电路结构；(b)设计风格#2 的电路结构

设计风格#2 的状态机模板

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY <ent_name>IS
   PORT (input: IN <data_type>;
        reset, clock: IN STD_LOGIC;
        output: OUT <data_type>);
END <ent_name>;
-------------------------------------------------
ARCHITECTURE <arch_name>OF<ent_name>IS
   TYPE states IS (state0, state1, state2, state3, ...);
   SIGNAL pr_state, nx_state: states;
   SIGNAL temp: <data_type>;
BEGIN
   -----Lower section: --------------------------
   PROCESS (reset, clock)
   BEGIN
      IF (reset = '1') THEN
         pr_state <= state0;
      ELSIF (clock'EVENT AND clock = '1') THEN
         output <= temp;
         pr_state <= nx_state;
      END IF;
   END PROCESS;
   -----Upper section: --------------------------
   PROCESS (pr_state)
```

```
    BEGIN
       CASE pr_state IS
          WHEN state0 =>
             temp <= <value>;
             IF (condition) THEN nx_state <= state1;
             ...
             END IF;
          WHEN state1 =>
             temp <= <value>;
             IF (condition) THEN nx_state <= state2;
             ...
             END IF;
          WHEN state2 =>
             temp <= <value>;
             IF (condition) THEN nx_state <= state3;
             ...
             END IF;
          ...
       END CASE;
    END PROCESS;
END <arch_name>;
```

比较设计风格#1和设计风格#2，只有一个差别，那就是引入了内部信号 temp。这个信号将输出结果存储起来，只有当所需的时钟边沿到来时才将它赋给输出端口。

例 8.3 简单的 FSM#2

再次考虑例 8.2。此时需要的是同步输出，输出结果只在时钟上升沿改变。由于这是一个米里型状态机，所以采用设计风格#2。

```
1  -------------------------------------------------
2  ENTITY simple_fsm IS
3     PORT (a, b, d, clk, rst: IN BIT;
4            x: OUT BIT);
5  END simple_fsm;
6  -------------------------------------------------
7  ARCHITECTURE simple_fsm OF simple_fsm IS
8     TYPE state IS (stateA, stateB);
9     SIGNAL pr_state, nx_state: state;
10    SIGNAL temp: BIT;
```

```
11 BEGIN
12    ---------Lower section: -----------------------
13    PROCESS (rst, clk)
14    BEGIN
15       IF (rst = '1') THEN
16          pr_state <= stateA;
17       ELSIF (clk'EVENT AND clk = '1') THEN
18          x <= temp;
19          pr_state <= nx_state;
20       END IF;
21    END PROCESS;
22    ----------Upper section: ----------------------
23    PROCESS (a, b, d, pr_state)
24    BEGIN
25       CASE pr_state IS
26          WHEN stateA =>
27             temp <= a;
28             IF (d = '1') THEN nx_state <= stateB;
29             ELSE nx_state <= stateA;
30             END IF;
31       WHEN stateB =>
32             temp <= b;
33             IF (d = '1') THEN nx_state <= stateA;
34             ELSE nx_state <= stateB;
35             END IF;
36       END CASE;
37    END PROCESS;
38 END simple_fsm;
39 -------------------------------------------------
```

查看编译器的报告文件，会发现整个电路生成了两个触发器，其中一个用来为状态机进行状态编码，另一个用来存储输出。

图 8.7 显示了其仿真结果。当信号被存储起来以后，其值在两个连续的时钟边沿之间必然保持不变。如果在这个时间间隔内输入发生变化，电路也许发现不了这个变化，即使能够发现，输出值的更新也会存在一个延迟。

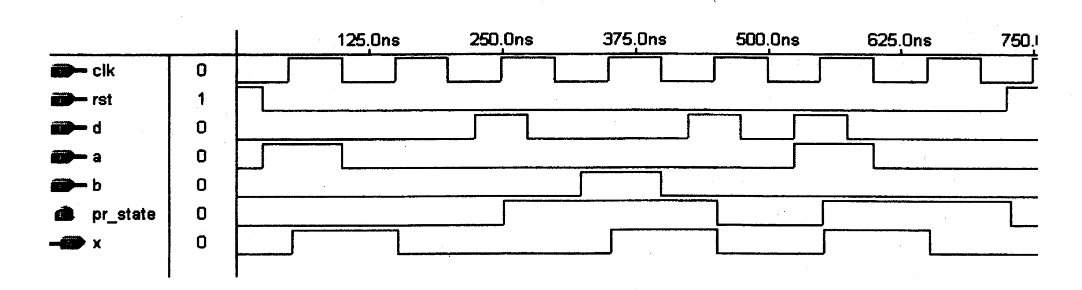


图 8.7 例 8.3 的仿真结果

例 8.4 序列检测器

下面要实现一个序列检测器电路，输入是一个串行位流，当出现序列"111"时，输出为'1'。这里必须考虑出现长连'1'的问题，也就是说，如果出现"…0111110…"，则输出就要保持连续 3 个时钟周期的'1'。

图 8.8 给出了这个状态机的状态转移图。这里有 4 个状态，命名为 zero，one，two 和 three，分别对应检测到的连续'1'的个数。下面给出的是使用设计风格#1 编写的代码。

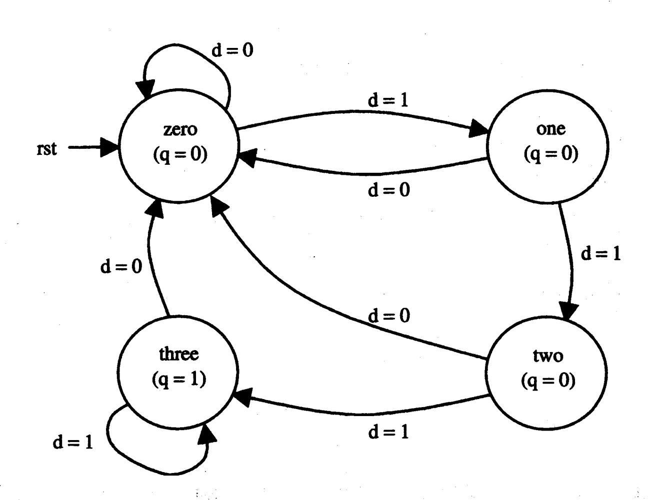


图 8.8 例 8.4 的状态转移图

```
1   --------------------------------------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   --------------------------------------------------
5   ENTITY string_detector IS
6      PORT (d, clk, rst: IN BIT;
```

```
7          q: OUT BIT);
8   END string_detector;
9   ---------------------------------------------------
10  ARCHITECTURE my_arch OF string_detector IS
11     TYPE state IS (zero, one, two, three);
12     SIGNAL pr_state, nx_state: state;
13  BEGIN
14     ----------------Lower section: ---------------
15     PROCESS (rst, clk)
16     BEGIN
17        IF (rst = '1') THEN
18           pr_state <= zero;
19        ELSIF (clk'EVENT AND clk = '1') THEN
20           pr_state <= nx_state;
21        END IF;
22     END PROCESS;
23     ----------------Upper section: ---------------
24     PROCESS (d, pr_state)
25     BEGIN
26        CASE pr_state IS
27           WHEN zero =>
28              q <= '0';
29              IF (d = '1') THEN nx_state <= one;
30              ELSE nx_state <= zero;
31              END IF;
32           WHEN one =>
33              q <= '0';
34              IF (d = '1') THEN nx_state <= two;
35              ELSE nx_state <= zero;
36              END IF;
37           WHEN two =>
38              q <= '0';
39              IF (d = '1') THEN nx_state <= three;
40              ELSE nx_state <= zero;
41              END IF;
42           WHEN three =>
43              q <= '1';
44              IF (d = '0') THEN nx_state <= zero;
45              ELSE nx_state <= three;
```

```
46              END IF;
47          END CASE;
48      END PROCESS;
49  END my_arch;
50  ---------------------------------------------------
```

在这个例子中，输出不取决于当前输入。这一点可从上面代码中的第 28 行、第 33 行、第 38 行和第 43 行看出来，即所有对 q 的赋值都是无条件的（不依赖于当前输入 d），因此输出是自动同步的（米里型状态机），所以没有必要采用设计风格#2。这个电路需要两个触发器，用来为状态机的 4 个状态编码，根据状态的变化来确定输出 q。

仿真结果如图 8.9 所示。可以看到，输入到电路中的数据序列 d = "011101100"，相应的输出为 q = "000100000"。

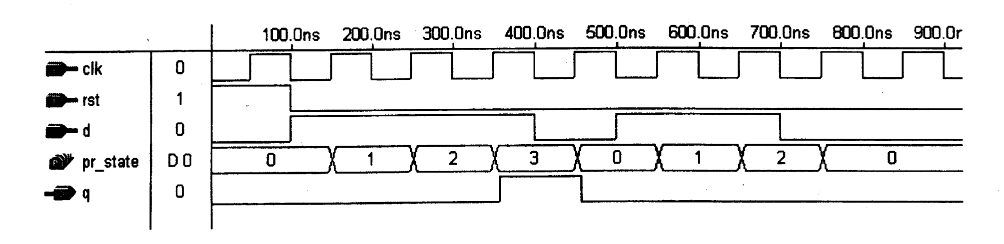
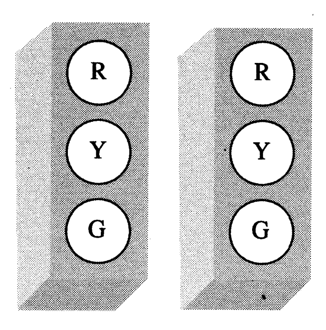


图 8.9 例 8.4 的仿真结果

例 8.5 交通灯控制器

前面曾经讲到，以建立状态机模型的方式设计数字控制器之类的电路是一种非常有效的设计方法。在下面这个实例中，将按照图 8.10 中的表格所总结的特征来设计交通灯控制器，具体分析如下：

1. 交通灯控制器有 3 个操作模式：正常模式、测试模式和检修模式。
2. 正常模式（Regular mode）：有 4 个状态，每个状态都具有可独立编程确定的时间参数（通过修改 CONSTANT）。
3. 测试模式（Test mode）：通过手动开关，允许使用很小的时间值来覆盖先前设定的时间参数，以便于在设备维护时进行快速的测试。这个值也是通过 CONSTANT 传递给电路的。
4. 检修模式（Standby mode）：一旦进入这个模式（例如传感器发现错误或通过手动开关进入），系统在两个方向上必须亮黄灯，并在相应控制信号有效的情况下保持此状态。
5. 假设使用的时钟频率为 60 Hz。



状态	操作模式		
	REGULAR	TEST	STANDBY
	Time	Time	Time
RG	timeRG (30s)	timeTEST (1s)	---
RY	timeRY (5s)	timeTEST (1s)	---
GR	timeGR (45s)	timeTEST (1s)	---
YR	timeYR (5s)	timeTEST (1s)	---
YY	---	---	不确定

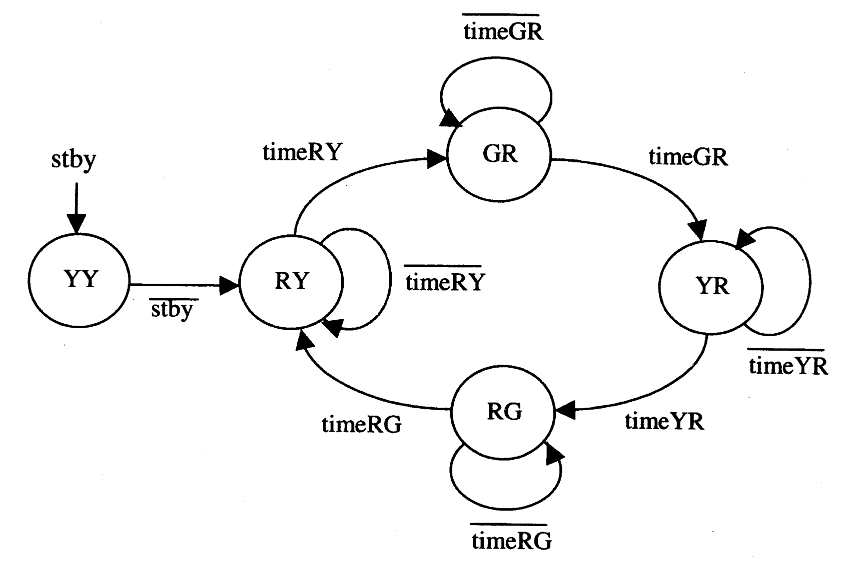


图 8.10　例 8.5 的设计规范和常规模式下的状态转移图

在这里使用的是设计风格#1，代码编写如下：

```
1  --------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY tlc IS
6     PORT (clk, stby, test: IN STD_LOGIC;
7              r1, r2, y1, y2, g1, g2: OUT STD_LOGIC);
8  END tlc;
9  --------------------------------------------------
10 ARCHITECTURE behavior OF tlc IS
11    CONSTANT timeMAX: INTEGER := 2700;
```

```
12    CONSTANT timeRG: INTEGER := 1800;
13    CONSTANT timeRY: INTEGER := 300;
14    CONSTANT timeGR: INTEGER := 2700;
15    CONSTANT timeYR: INTEGER := 300;
16    CONSTANT timeTEST: INTEGER := 60;
17    TYPE state IS (RG, RY, GR, YR, YY);
18    SIGNAL pr_state, nx_state: state;
19    SIGNAL time: INTEGER RANGE 0 TO timeMAX;
20 BEGIN
21    --------Lower section of state machine:-------
22    PROCESS (clk, stby)
23       VARIABLE count: INTEGER RANGE 0 TO timeMAX;
24    BEGIN
25       IF (stby = '1') THEN
26          pr_state <= YY;
27          count := 0;
28       ELSIF (clk'EVENT AND clk = '1') THEN
29          count := count+1;
30          IF (count=time) THEN
31             pr_state <= nx_state;
32             count := 0;
33          END IF;
34       END IF;
35    END PROCESS;
36    --------Upper section of state machine: ------
37    PROCESS (pr_state, test)
38    BEGIN
39       CASE pr_state IS
40          WHEN RG =>
41             r1 <= '1'; r2 <= '0'; y1 <= '0'; y2 <= '0'; g1 <= '0'; g2 <= '1';
42             nx_state <= RY;
43             IF (test = '0') THEN time <= timeRG;
44             ELSE time <= timeTEST;
45             END IF;
46          WHEN RY =>
47             r1 <= '1'; r2 <= '0'; y1 <= '0'; y2 <= '1'; g1 <= '0'; g2 <= '0';
48             nx_state <= GR;
49             IF (test = '0') THEN time <= timeRY;
50             ELSE time <= timeTEST;
```

```
51                END IF;
52            WHEN GR =>
53                r1 <= '0'; r2 <= '1'; y1 <= '0'; y2 <= '0'; g1 <= '1'; g2 <= '0';
54                nx_state <= YR;
55                IF (test = '0') THEN time <= timeGR;
56                ELSE time <= timeTEST;
57                END IF;
58            WHEN YR =>
59                r1 <= '0'; r2 <= '1'; y1 <= '1'; y2 <= '0'; g1 <= '0'; g2 <= '0';
60                nx_state <= RG;
61                IF (test = '0') THEN time <= timeYR;
62                ELSE time <= timeTEST;
63                END IF;
64            WHEN YY =>
65                r1 <= '0'; r2 <= '0'; y1 <= '1'; y2 <= '1'; g1 <= '0'; g2 <= '0';
66                nx_state <= RY;
67        END CASE;
68     END PROCESS;
69 END behavior;
70 --------------------------------------------------
```

实现这个电路预计需要的寄存器数量是 15 个。3 个用来存储 pr_state（状态机有 5 个状态，所以需要 3 个寄存器对它进行编码），另外 12 个供计数器使用（这里采用 12 位的计数器，因为需要的最大计数值为 MAX = 2700）。

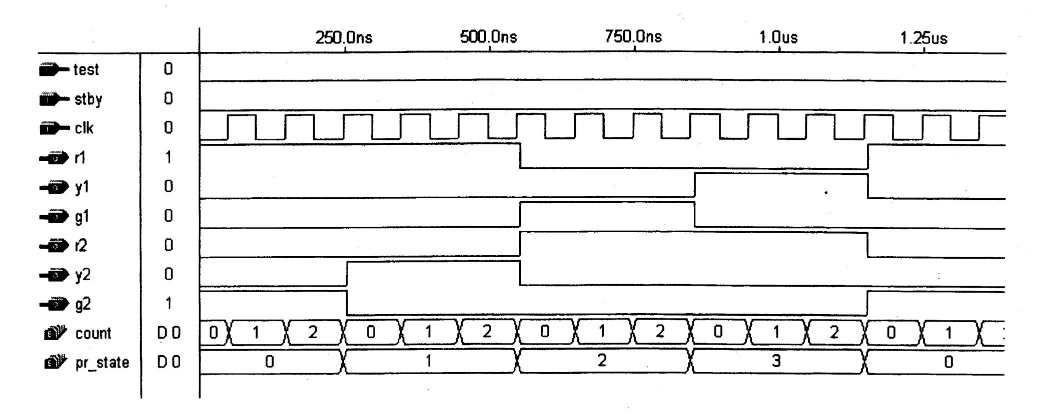
仿真结果如图 8.11 所示。为了使仿真结果能够在图形中充分反映出来，这里采用了较小的时间值。除了测试模式下的 CONSTANT 为 1 以外，其他模式下的 CONSTANT 都等于 3。因此，在正常模式下，状态每 3 个时钟周期变化一次，在测试模式下每个时钟周期变化一次。图 8.11 的前两个图仿真了这两种模式，第三幅图仿真了检修模式下的电路工作情况。正如期望的一样，stby 是异步的，并且比 test 具有更高的优先级，这样在 stby 有效时系统会一直停留在 YY 状态（state4）。另一方面，test 信号是同步的，状态机在检测到 test 有效以后，立刻进入测试状态，不需要等待当前状态完成，这在第二幅图中可以看出来。

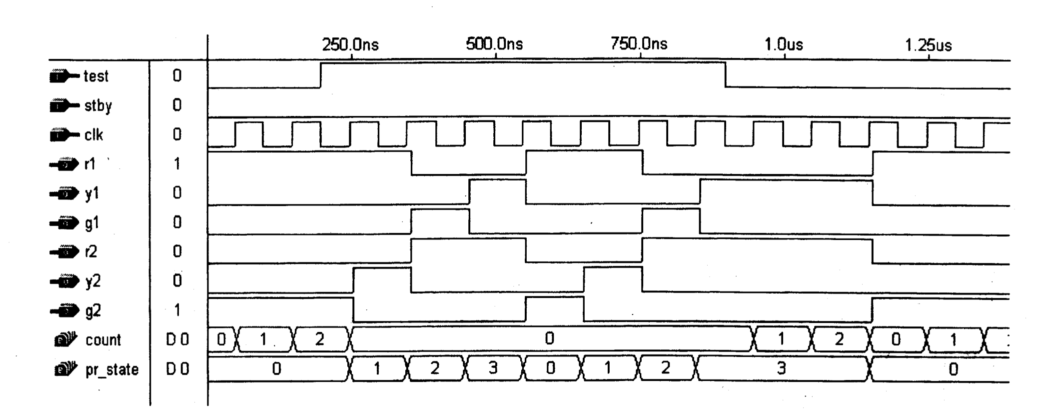
例 8.6 信号发生器

在本例题中将设计产生如图 8.12（a）所示波形的电路，该电路只有一个输入时钟信号。需要注意的是，输出信号在时钟的两个边沿（上升沿和下降沿）都会发生变化。

为了解决电路在时钟的两个边沿都工作的问题，一个做法是使用两个状态机，一个只在时钟上升沿工作，而另一个只在时钟下降沿工作（见 6.9 节），它们分别产生如图 8.12（b）中 out1 和

out2 所示的波形。out1 和 out2 相“与”后可以得到所期望的输出信号。由于这个电路没有除时钟以外的其他输入信号，所以输出只有当时钟改变时才发生变化（同步输出）。





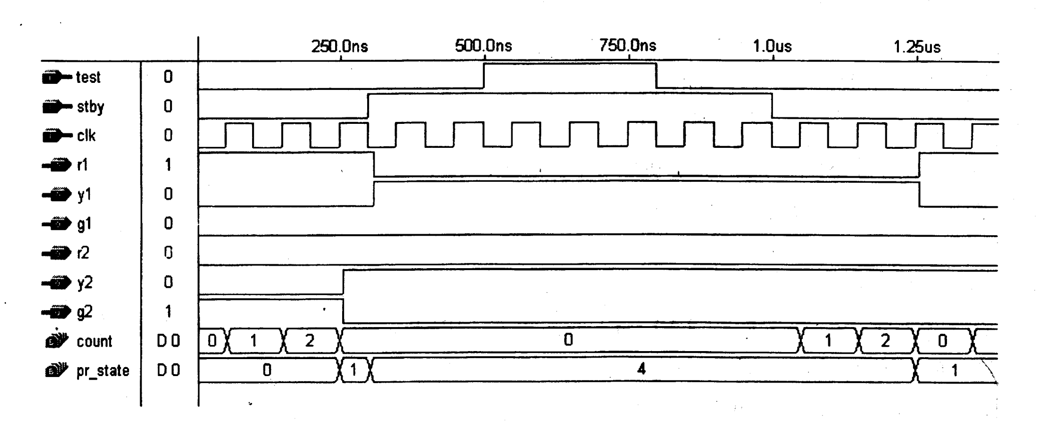


图 8.11 例 8.5 的仿真结果

```
1  --------------------------------------------------
2  ENTITY signal_gen IS
3     PORT (clk: IN BIT;
4           outp: OUT BIT);
5  END signal_gen;
6  --------------------------------------------------
7  ARCHITECTURE fsm OF signal_gen IS
8     TYPE state IS (one, two, three);
9     SIGNAL pr_state1, nx_state1: state;
10    SIGNAL pr_state2, nx_state2: state;
11    SIGNAL out1, out2: BIT;
12 BEGIN
13 -----------Lower section of machine #1: ---------
14 PROCESS (clk)
15 BEGIN
16    IF (clk'EVENT AND clk = '1') THEN
17       pr_state1 <= nx_state1;
18    END IF;
19 END PROCESS;
20 ---------- Lower section of machine #2: ----------
21 PROCESS (clk)
22 BEGIN
23    IF (clk'EVENT AND clk = '0') THEN
24       pr_state2 <= nx_state2;
25    END IF;
26 END PROCESS;
27 ----------Upper section of machine #1: -----------
28 PROCESS (pr_state1)
29 BEGIN
30    CASE pr_state1 IS
31       WHEN one =>
32          out1 <= '0';
33          nx_state1 <= two;
34       WHEN two =>
35          out1 <= '1';
36          nx_state1 <= three;
37       WHEN three =>
38          out1 <= '1';
```

```
39            nx_state1 <= one;
40         END CASE;
41     END PROCESS;
42     -------- Upper section of machine #2: ---------
43     PROCESS (pr_state2)
44     BEGIN
45        CASE pr_state2 IS
46           WHEN one =>
47              out2 <= '1';
48              nx_state2 <= two;
49           WHEN two =>
50              out2 <= '0';
51              nx_state2 <= three;
52           WHEN three =>
53              out2 <= '1';
54              nx_state2 <= one;
55        END CASE;
56     END PROCESS;
57     outp <= out1 AND out2;
58 END fsm;
59 -------------------------------------------------
```

（a）

（b）

图 8.12 例 8.6 的波形。（a）outp 由 clk 产生；（b）outp = out1 AND out2

上面的代码综合后的仿真结果如图 8.13 所示。

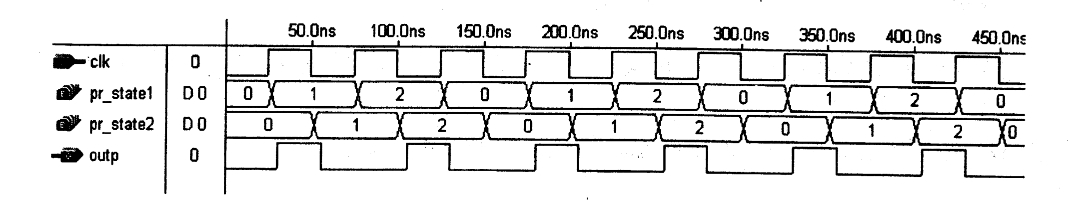


图 8.13　例 8.6 的仿真结果

8.4　状态机编码风格：二进制编码和独热编码

我们有多种方式对状态机的状态进行编码。默认的方式是进行二进制编码。它的优点是需要的寄存器数量最少。在这种情况下，有 n 个寄存器就可以对 2^n 个状态进行编码。与其他编码方式相比，这种编码方式的缺点就是需要更多的外部辅助逻辑，并且速度较慢。

另一种极端的编码方式就是独热（OneHot）编码。在这种编码方式中，每个状态都需要一个寄存器。因此，它需要的寄存器数量最多。在这种情况下，对 n 个状态编码就需要 n 个寄存器，但这种方法需要最少的辅助逻辑并具有最快的速度。

一种介于上面两者之间的编码方式是双热编码方式。在这种编码方式下，每一次状态变化会带来两个位的跳变，因此 n 个寄存器可以实现对 $n(n-1)/2$ 个状态进行编码。

在寄存器资源比较多的情况下建议用独热编码，譬如在 FPGA 中可以使用独热编码，而在 ASIC 中二进制编码方式是最好的。

我们以一个具有 8 个状态的状态机为例进行对比说明。表 8.1 给出了采用不同方式进行编码后得到的状态编码表，从中可以看出每种编码方式需要的寄存器数量。

表 8.1　8 状态 FSM 的状态编码

状态	编码风格		
	二进制	双热	独热
state0	000	00011	00000001
state1	001	00101	00000010
state2	010	01001	00000100
state3	011	10001	00001000
state4	100	00110	00010000
state5	101	01010	00100000
state6	110	10010	01000000
state7	111	01100	10000000

8.5 习题

下面每个习题都需要进行综合与仿真，并至少要验证电路实际使用的寄存器数量以及所实现电路的功能是否正确。

8.1 FSM。编写实现如图 P8.1 所示状态转移关系的 VHDL 代码。

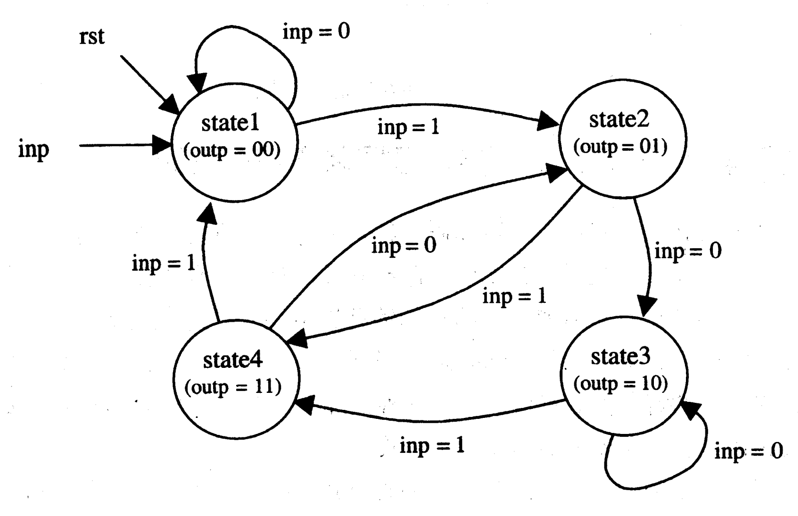


图 P8.1

8.2 信号发生器#1。使用有限状态机设计一个电路，它可以产生图 P8.2 所描述的两个信号（out1 和 out2），该电路只有一个输入时钟信号 clk。out1 和 out2 都是周期信号，且周期长度相同。在两个信号中，一个在靠近 clk 的上升沿触发，另一个在 clk 的两个边沿上都会发生变化。

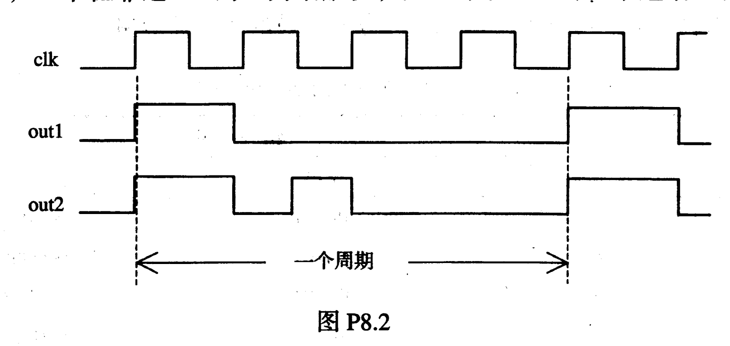


图 P8.2

8.3 信号发生器#2。设计一个有限状态机，它可以实现如图 P8.3 描述的两个信号：UP 和 DOWN。这两个信号被两个输入信号 GO 和 STOP 控制。当 GO 从'0'跳变到'1'时，UP 在 10 ms 后必须也跳变成'1'。如果 GO 再跳变回'0'，UP 应立刻跳变成'0'，同时 DOWN 在 10 ms 后跳变为'1'，DOWN 在 GO 跳变为'1'后立刻返回'0'。如果输入 STOP 有效，则两者都要无条件地立即变为'0'。这里假设时钟频率为 10 kHz。

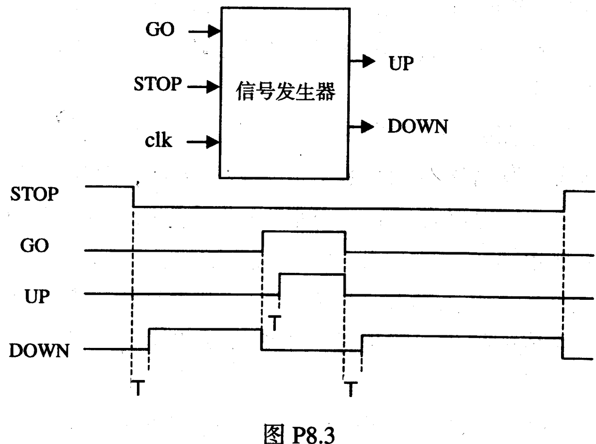
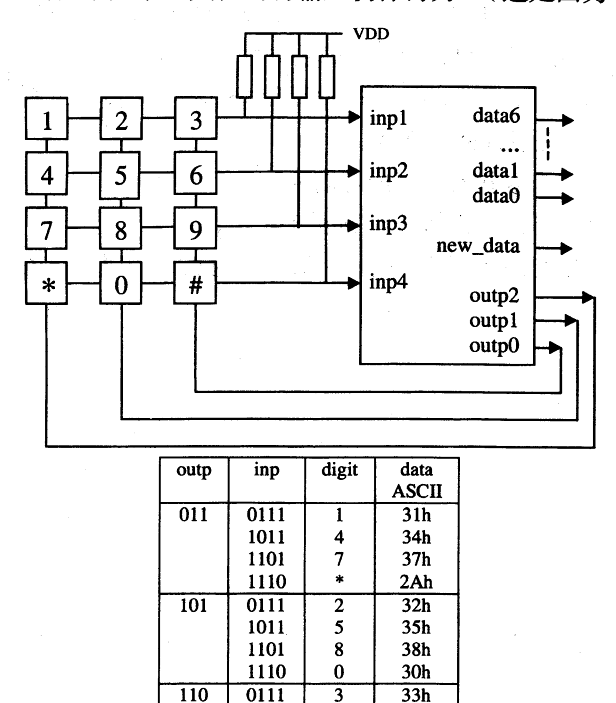


图 P8.3

8.4　按键防抖动电路和编码器。图 P8.4 是键盘编码和防抖动控制电路。我们通常采用扫描和轮询的方法来检测对键盘的敲击，这种方法可以减少键盘与主电路之间的连线数量。该电路每次将 outp0，outp1 和 outp2 中的一个置'0'，然后依次读入 inp1，inp2，inp3 和 inp4。如果某个键被按下，则相应的行输入为'0'，而其他的行输入仍保持为'1'(这是因为每一行都有上拉电阻)。



outp	inp	digit	data ASCII
011	0111	1	31h
	1011	4	34h
	1101	7	37h
	1110	*	2Ah
101	0111	2	32h
	1011	5	35h
	1101	8	38h
	1110	0	30h
110	0111	3	33h
	1011	6	36h
	1101	9	39h
	1110	#	23h

图 P8.4

编码功能：每个数字必须被编为 ASCII 码（参见图 P8.4 中的表格）。编码电路通过将 new_data 置为'1'，表示电路检测到一次键盘敲击并输出有效编码。这种做法可以避免长期按下一个键而出现一长串相同的数字。

防抖动电路：机械开关存在的一个问题是开关抖动。开关抖动是指开关在形成最终稳定接触之前出现的不稳定状态，其持续时间一般为几毫秒。因此，时钟频率的选择相当重要。试选择一个适当的时钟频率，使电路在 5 ms 时间内至少能进行 3 次读操作，只有当在 5 ms 内某个键的所有读出结果都相同时，new_data 才被置为'1'。

8.5 交通灯控制器。运用你的综合工具及 CPLD/FPGA 开发环境，实现例 8.5 的交通灯控制器电路。分析综合工具所生成的报告，找出输入引脚和输出引脚，然后在电路板上进行下面的物理连接。

（a）将 60 Hz 的方波信号（信号发生器产生）以合适的电平连接到 clk 引脚。

（b）将 VDD/GND 开关连接到 stby 引脚上。

（c）将 VDD/GND 开关连接到 test 引脚上。

（d）选择一个电阻（阻值在 330~1000 Ω之间），将它的一端与 r1 连接，另一端与一个红色发光二极管（LED）的正极连接，LED 的负极接地。

（e）选择一个黄色 LED，采用与（d）相同的方法，连接到 y1 上。

（f）选择一个绿色 LED，采用与（d）相同的方法，连接到 g1 上。

最后，可以在 CPLD/FPGA 开发环境中通过编程电缆将这个设计下载到可编程器件中，并对整个电路进行测试，验证设计的正确性。

8.6 信号发生器#3。不使用有限状态机完成习题 8.2。

8.7 信号发生器#4。不使用有限状态机完成例题 8.6。

关于更复杂的有限状态机设计，可参见第 9 章的习题 9.3、习题 9.4 和习题 9.6。

第 9 章　典型电路设计分析

前面的章节以例题的方式使用 VHDL 代码实现了一系列简单电路，对于每个例题都给出了以下内容：

- 给出了顶层电路图及相关说明
- 在必要时对数字电路的基本概念进行了回顾
- 给出了 VHDL 代码
- 给出仿真结果
- 进行了必要的注释

本章是第一部分的最后一章，将分析一系列设计实例。这些例子和迄今为止列举的其他例题一样，针对的都是功能相对简单和独立的电路（也就是说，一段代码就可以实现一个完整的电路）。在第二部分的最后一章，我们同样会给出典型的系统级电路设计的例题作为该部分学习的总结。本章将分析以下例题：

- 桶形移位寄存器
- 有符号数和无符号数比较器
- 逐级进位和超前进位加法器
- 定点除法器
- 自动售货机控制器
- 串行数据接收器
- 并/串转换器
- 7 段数码显示器的使用
- 信号发生器
- 存储器

最后，本章还包括了一系列习题。

本书的所有例题都列举在 1.5 节中，以便于根据需要进行查找。

9.1　桶形移位寄存器

桶形移位寄存器的电路结构如图 9.1 所示。它的输入是位宽为 8 的矢量，输出是将输入移位后的结果，移动的位数由输入的 shift 值（取值范围是 0~7）决定。该电路包括 3 个独立的桶形移

位寄存器，每个都和例 6.9 中给出的电路结构相似。在图 9.1 左侧的第一级寄存器中，只有 1 个'0'连接到左下角的一个复用器上。而第二级有 2 个'0'，第三级有 3 个'0'。对于位宽更高的矢量，将采用这种逐级加倍插入'0'的方法来构造桶形移位寄存器。如果 shift = "001"，则只有第一级进行移位；而当 shift = "111"时，所有的三级都要进行移位操作。

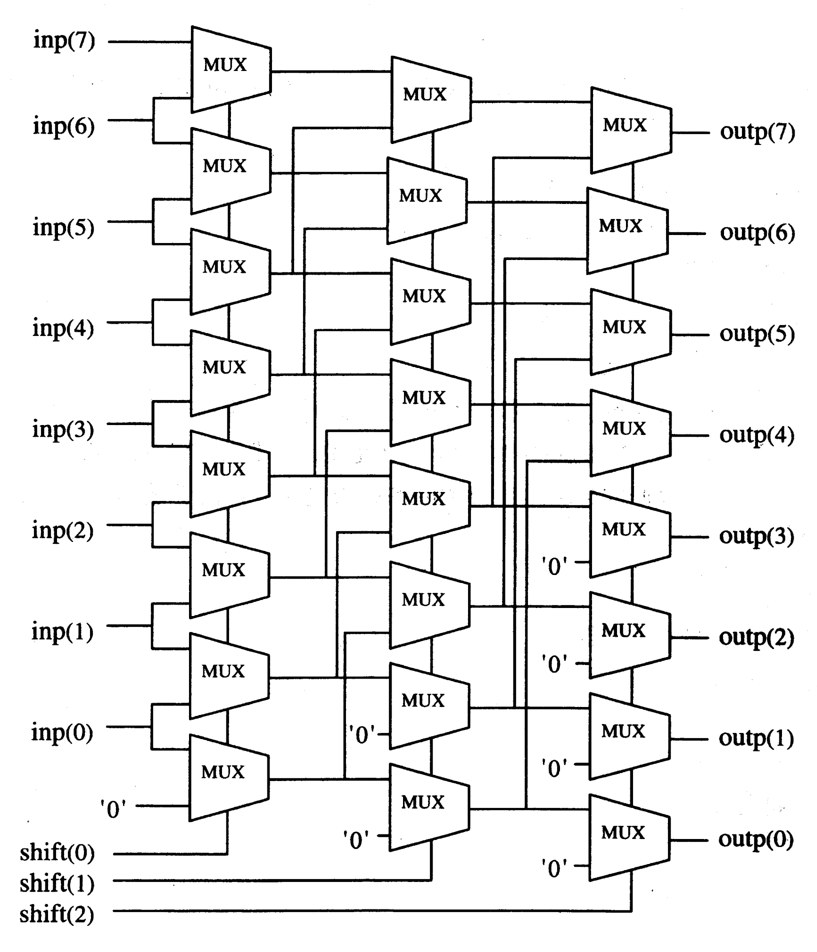


图 9.1　桶形移位寄存器

```
1  ---------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY barrel IS
6     PORT (inp: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7           shift: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
8           outp: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
```

```
9  END barrel;
10 --------------------------------------------------
11 ARCHITECTURE behavior OF barrel IS
12 BEGIN
13    PROCESS (inp, shift)
14       VARIABLE temp1: STD_LOGIC_VECTOR(7 DOWNTO 0);
15       VARIABLE temp2: STD_LOGIC_VECTOR(7 DOWNTO 0);
16    BEGIN
17       ----------------1st shifter---------------
18       IF (shift(0) = '0') THEN
19          temp1 := inp;
20       ELSE
21          temp1(0) := '0';
22          FOR i IN 1 TO inp'HIGH LOOP
23             temp1(i) := inp(i-1);
24          END LOOP;
25       END IF;
26       ---------------2nd shifter---------------
27       IF (shift(1) = '0') THEN
28          temp2 := temp1;
29       ELSE
30          FOR i IN 0 TO 1 LOOP
31             temp2(i) := '0';
32          END LOOP;
33          FOR i IN 2 TO inp'HIGH LOOP
34             temp2(i) := temp1(i-2);
35          END LOOP;
36       END IF;
37       ----------------3rd shifter-------------
38       IF (shift(2) = '0') THEN
39          outp <= temp2;
40       ELSE
41          FOR i IN 0 TO 3 LOOP
42             outp(i) <= '0';
43          END LOOP;
44          FOR i IN 4 TO inp'HIGH LOOP
45             outp(i) <= temp2(i-4);
46          END LOOP;
47       END IF;
```

```
48     END PROCESS;
49 END behavior;
50 ----------------------------------------------------
```

上面给出了该电路的 VHDL 代码。仿真结果由图 9.2 给出。在图 9.2 中可以看到，当 shift ='0'时，输出和输入相同。通过简单分析可以发现，在移位操作中，如果没有'1'被移出，那么移位后的结果等于将输入值乘以 2（移位 1 次）、乘以 4（移位 2 次）和乘以 8（移位 3 次）。

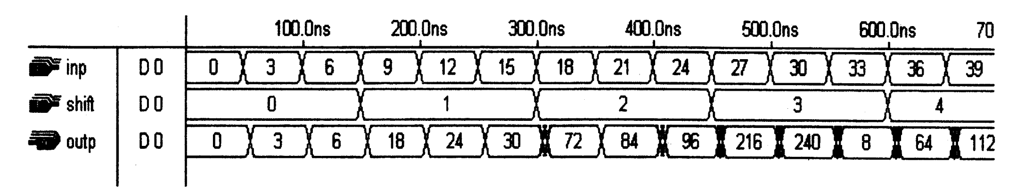


图 9.2　图 9.1 电路的仿真结果

9.2　有符号数比较器和无符号数比较器

图 9.3 给出了比较器的顶层框图。参加比较的矢量宽度均为 n+1。该电路的 3 个输出为 x1（当 a > b 时输出为'1'），x2（当 a = b 时输出为'1'）和 x3（当 a<b 时输出为'1'）。这里给出了 3 段代码，第一段代码中 a 和 b 是有符号数，后两段代码中 a 和 b 是无符号数。对于 3 种实现方法都给出了仿真结果。

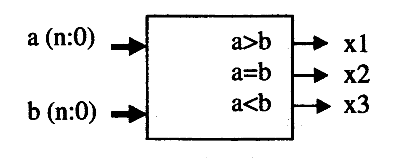


图 9.3　比较器

有符号数比较器

需要注意的是，在下面的代码中使用了 std_logic_arith 包集，它对于有符号和无符号类型数据的操作非常重要（第 8 行声明 a 和 b 是有符号数）。

```
1 ---------------Signed Comparator: ---------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_arith.all; --necessary!
5 ---------------------------------------------
6 ENTITY comparator IS
7     GENERIC(n: INTEGER := 7);
```

```
8      PORT (a, b: IN SIGNED(n DOWNTO 0);
9             x1, x2, x3: OUT STD_LOGIC);
10 END comparator;
11 --------------------------------------------------
12 ARCHITECTURE signed OF comparator IS
13 BEGIN
14      x1 <= '1'WHEN a>b ELSE '0';
15      x2 <= '1'WHEN a=b ELSE '0';
16      x3 <= '1'WHEN a<b ELSE '0';
17 END signed;
18 --------------------------------------------------
```

图 9.4 给出了仿真结果。可以看到，127 大于 0，而 128 和 255 都小于 0（有符号数 127 的二进制补码就是十进制中的 127 本身，而有符号数 128 的二进制补码是十进制中的–128。同样，255 实际上是–1）。

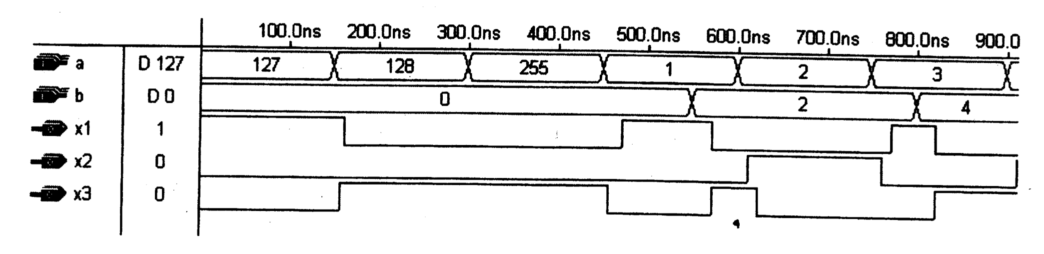


图 9.4　图 9.3 中有符号数比较器的仿真结果

无符号数比较器#1

下面的 VHDL 代码是刚才的代码（有符号数比较器）的复制品。再次注意 std_logic_arith 包集的存在，这个包集对于无符号（或有符号）数据的操作非常重要（在第 8 行中声明了 a 和 b 是无符号的）。

```
1  -----------Unsigned Comparator#1: ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all; --necessary!
5  --------------------------------------------------
6  ENTITY comparator IS
7     GENERIC(n: INTEGER := 7);
8     PORT (a, b: IN UNSIGNED(n DOWNTO 0);
9             x1, x2, x3: OUT STD_LOGIC);
10 END comparator;
```

```
11 ------------------------------------------------
12 ARCHITECTURE unsigned OF comparator IS
13 BEGIN
14     x1 <= '1'WHEN a > b ELSE '0';
15     x2 <= '1'WHEN a = b ELSE '0';
16     x3 <= '1'WHEN a < b ELSE '0';
17 END unsigned;
18 ------------------------------------------------
```

无符号数比较器#2

无符号数比较器同样可以对 STD_LOGIC_VECTORS 类型的数据进行比较，这样就可以不必声明 std_logic_arith 包集了。具体的描述方法由下面的代码给出：

```
1  ----------- Unsigned Comparator#2: --------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY comparator IS
6     GENERIC(n: INTEGER := 7);
7     PORT (a, b: IN STD_LOGIC_VECTOR(n DOWNTO 0);
8           x1, x2, x3: OUT STD_LOGIC);
9  END comparator;
10 ------------------------------------------------
11 ARCHITECTURE unsigned OF comparator IS
12 BEGIN
13     x1 <= '1'WHEN a > b ELSE '0';
14     x2 <= '1'WHEN a = b ELSE '0';
15     x3 <= '1'WHEN a < b ELSE '0';
16 END unsigned;
17 ------------------------------------------------
```

无符号比较器的仿真结果如图 9.5 所示。与图 9.4 比较可以发现，这时的 128 和 255 的确比 0 大。

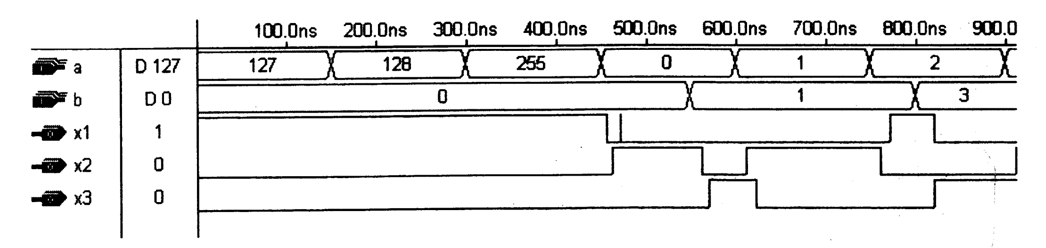


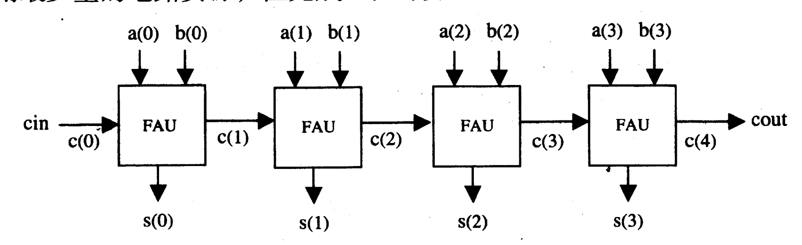
图 9.5　无符号数比较器的仿真结果

9.3 逐级进位和超前进位加法器

逐级进位和超前进位是设计加法器的两种经典方法。前者的优势是需要较少的硬件资源，而后者则速度较快。下面将介绍这两种方法。

逐级进位加法器

图 9.6 给出了一个 4 位无符号数的逐级进位加法器。对每一位都使用了全加器 FAU（见 1.4 节），同时给出了全加器的真值表。真值表中的 a 和 b 是输入位，cin 是进位输入位，s 是求和的结果，cout 是进位输出位。当输入位有奇数个'1'时，s 必定是'1'，而当有两个或更多的输入为'1'时，cout 必定是'1'。注意，图 9.6 中的每个全加器的输出结果都依赖于前一级产生的进位。这种方法可以占用最少量的电路资源，但完成一次计算所需的时延显然较大。



a b	cin	s	cout
0 0	0	0	0
0 1	0	1	0
1 0	0	1	0
1 1	0	0	1
0 0	1	1	0
0 1	1	0	1
1 0	1	0	1
1 1	1	1	1

图 9.6　4 位逐级进位加法器和 1 位全加器的真值表

基于图 9.6 给出的真值表，可以写出如下的逻辑表达式：

s = a XOR b XOR cin

cout = (a AND b) OR (a AND cin) OR (b AND cin)

可见，采用 VHDL 实现带进位的加法器可以采用十分直接的方法。下面的全加器的输入位宽可以是任意的，位宽由第 5 行的 GENERIC 语句确定。下面的代码综合后的仿真结果如图 9.7 所示。

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  ---------------------------------------------
4  ENTITY adder_cripple IS
5     GENERIC(n: INTEGER := 4);
6     PORT (a, b: IN STD_LOGIC_VECTOR(n-1 DOWNTO 0);
7           cin: IN STD_LOGIC;
8           s: OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0);
9           cout: OUT STD_LOGIC);
10 END adder_cripple;
11 ---------------------------------------------
12 ARCHITECTURE adder OF adder_cripple IS
13    SIGNAL c: STD_LOGIC_VECTOR(n DOWNTO 0);
```

```
14 BEGIN
15    c(0) <= cin;
16    G1: FOR i IN 0 TO n-1 GENERATE
17        s(i) <= a(i) XOR b(i) XOR c(i);
18        c(i+1) <= (a(i)AND b(i))OR
19                  (a(i)AND c(i))OR
20                  (b(i)AND c(i));
21    END GENERATE;
22    cout <= c(n);
23 END adder;
24 -------------------------------------------------
```

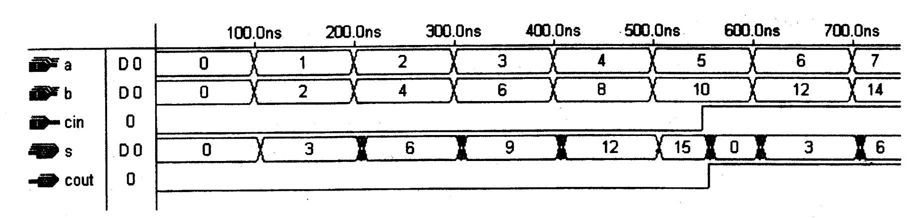


图 9.7　逐级进位加法器的仿真结果

预定义的"+"操作符

前面已经知道，加法器可以直接使用加法运算操作符来实现（见 4.1 节）。在使用"+"操作符的情况下，综合工具一般会采用逐级进位加法器来实现目标电路。如果不希望采用这种方法，则必须在代码描述上清晰地体现出来。

超前进位加法器

图 9.8 给出了 4 位超前进位加法器的电路结构图。电路实现时需要两个非常重要的中间信号：generate 和 propagate。假设两个输入位是 a 和 b，则 generate（下面用 g 来代替）和 propagate（下面用 p 来代替）信号定义如下：

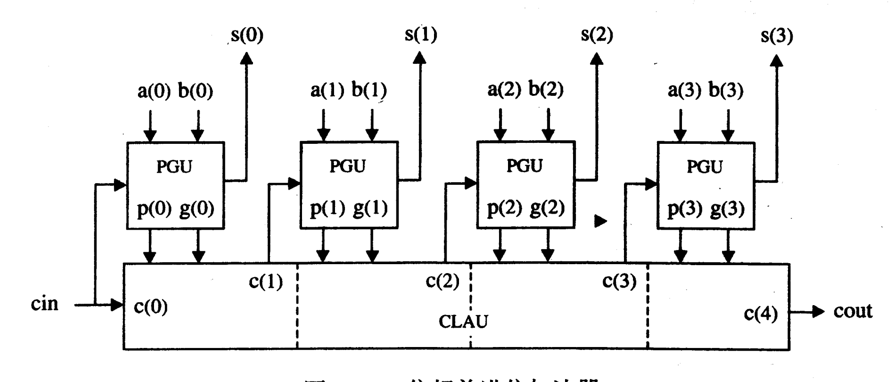


图 9.8　4 位超前进位加法器

g = a AND b

p = a XOR b

需要注意的是，这两个信号与进位无关，只根据当前的输入就可以计算。

现在假设有两个输入矢量：a = a(n−1)a(n−2)···a(1)a(0)和 b = b(n−1)b(n−2)···b(1)b(0)，那么相应的 generate 矢量为 g = g(n−1)g(n−2)···g(1)g(0)，相应的 propagate 矢量为 p = p(n−1)p(n−2)···p(1)p(0)。其中

g(j) = a(j) AND b(j)

p(j) = a(j) XOR b(j)

现在假设进位矢量为 c = c(n−1)c(n−2)···c(1)c(0)。进位可以由 g 和 p 按照下面的方法计算得到：

c(0) = cin

c(1) = c(0)p(0)+g(0)

c(2) = c(0)p(0)p(1)+g(0)p(1)+g(1)

c(3) = c(0)p(0)p(1)p(2)+g(0)p(1)p(2)+g(1)p(2)+g(2)，等等

与逐级进位加法器不同的是，上面的进位表达式中不包括前面各级的进位输出，都可以进行独立计算。也就是说，上面的表达式没有一个取决于前级进位输出的计算结果，这就是该电路执行速度快的根本原因。另一方面，这种方式会使硬件的复杂程度增加，所以只能用于实现位数不是很多的加法器（如 4 位）。将类似于 4 位超前进位加法器的小规模电路组合起来，可以实现位宽更高的超前进位加法器。

现在要实现图 9.8 所示的加法器就相对简单些了。PGU（产生和传递单元）计算 g 和 p（共需要 4 个），CLAU（超前进位单元）计算进位结果，就可以很方便地得到计算结果 s 了。

如果要构建较大的超前进位加法器，图 9.8 中的 CLAU 必须与组进位器（GP）和组产生器（GG）相连，这在图 9.8 中被忽略了，因为图中实现的只是 4 位全加器。

```
1  ---------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  ENTITY CLA_Adder IS
6     PORT (a, b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
7           cin: IN STD_LOGIC;
8           s: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
9           cout: OUT STD_LOGIC);
10 END CLA_Adder;
11 ---------------------------------------------------
12 ARCHITECTURE CLA_Adder OF CLA_Adder IS
13    SIGNAL c: STD_LOGIC_VECTOR(4 DOWNTO 0);
```

```
14     SIGNAL p: STD_LOGIC_VECTOR(3 DOWNTO 0);
15     SIGNAL g: STD_LOGIC_VECTOR(3 DOWNTO 0);
16 BEGIN
17     ----PGU: -------------------------------------
18     G1: FOR i IN 0 TO 3 GENERATE
19        p(i) <= a(i)XOR b(i);
20        g(i) <= a(i)AND b(i);
21        s(i) <= p(i)XOR c(i);
22     END GENERATE;
23     --------CLAU: --------------------------------
24     c(0) <= cin;
25     c(1) <= (cin AND p(0))OR
26                g(0);
27     c(2) <= (cin AND p(0) AND p(1)) OR
28                (g(0) AND p(1)) OR
29                g(1);
30     c(3) <= (cin AND p(0) AND p(1)AND p(2)) OR
31                (g(0) AND p(1)AND p(2)) OR
32                (g(1)AND p(2))OR
33                g(2);
34     c(4) <= (cin AND p(0) AND p(1)AND p(2)AND p(3)) OR
35                (g(0) AND p(1)AND p(2)AND p(3)) OR
36                (g(1)AND p(2)AND p(3))OR
37                (g(2)AND p(3))OR
38                g(3);
39     cout <= c(4);
40 END CLA_Adder;
41 -------------------------------------------------
```

上面的代码经过电路综合后的仿真结果与图 9.7 中逐级进位加法器的仿真结果相似。

9.4 定点除法

通过第 4 章可以知道，预定义运算符 "/" 只能进行 2^n 类型的除法运算，其运算本质实际上就是移位操作。本节将讨论实现一般意义上的除法的具体方式，这里的除数和被除数可以是任意整数。这里先介绍除法的运算法则，然后给出两段 VHDL 代码以及相应的仿真结果。

除法电路的算法

假设要计算 y = a/b，其中 a，b 和 y 有相同的位数(n+1 位)。参考图 9.9，假设 a = "1011"，b = "0011"，

我们希望得到的除法运算结果是 y = "0011"和余数"0010"。参见图 9.9 中与 b_inp 相关的一栏。在运算前，我们首先构建一个长度为 2n+1 的信号，这个信号的高 n+1 位与 b 完全相同，b_inp(i)是 b 简单地向左移 i 位（图 9.9 中与 b 相关的输入一栏中有下划线的部分）得到的。

索引 (i)	与 a 相关的输入（a_inp）	比较	与 b 相关的输入（b_inp）	商	操作
3	1011	<	0011000	0	none
2	1011	<	0001100	0	none
1	1011	>	0000110	1	a_inp(i)–b_inp(i)
0	0101	>	0000011	1	a_inp(i)–b_inp(i)
	0010 (rem)				

图 9.9　除法算法

下面分析商的计算方法。首先从图 9.9 的顶部开始对 a_inp(i)和 b_inp(i)进行比较。如果前者大于或等于后者，则 y(i)为'1'，然后将 a_inp(i)减去 b_inp(i)；否则 y(i) = '0'，再进入下一行。当 n+1 次迭代之后，计算完成，剩下的 a_inp 就是余数。

显然，如果在 a_inp 和 b_inp 之间进行减法运算，则 a_inp 的位数不能比 b_inp 少，所以 a_inp 的实际长度必须增加，此时在 a_inp 的左边填充'0'就可以做到（在图 9.9 中没有写出 a_inp 左侧的 3 个'0'）。

下面介绍另一种除法算法。让 b 乘以 2^n 相当于将 b 向左移 n 位，所以新的 b 矢量比原矢量长 n 位。如果 a 比新的 b 更大，则 y(n)取值为'1'，然后将 a 减去 b（移位后的新值）；否则取 y='0'。此后进行新的迭代。将 b 乘以 2^{n-1} 相当于将 b 向左移 n–1 位，或将刚才计算时用到的值向右移 1 位。然后再将它和 a 比较，重复前面的步骤来决定 y(n–1)是取'0'还是'1'。整个计算过程需要循环进行 n+1 次上述的操作。

VHDL 除法器

下面是两段进行除法运算的 VHDL 代码。它们采用的都是顺序代码。第一个使用的是 IF 语句，另一个使用的是 LOOP 语句加上 IF 语句。第一种方案是按部就班地进行编码，所以可以看出代码与上面的算法是明显对应的。第二种方案更简捷一些，并且更通用（位宽参数 n 是通过第 6 行的 GENERIC 来声明的）。在这两种实现方案中，由于被除数不能为'0'，所以在下面的代码中都要检测 b 是否为'0'。仿真结果见图 9.10。

```
1  -------------方案 1: step-by-step----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY divider IS
6     PORT (a, b: IN INTEGER RANGE 0 TO 15;
```

```
7            y: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
8            rest: OUT INTEGER RANGE 0 TO 15;
9            err: OUT STD_LOGIC);
10 END divider;
11 -------------------------------------------------
12 ARCHITECTURE rtl OF divider IS
13 BEGIN
14     PROCESS (a, b)
15        VARIABLE temp1: INTEGER RANGE 0 TO 15;
16        VARIABLE temp2: INTEGER RANGE 0 TO 15;
17     BEGIN
18        --------------Error and initialization: ---
19        temp1 := a;
20        temp2 := b;
21        IF (b=0) THEN err <= '1';
22        ELSE err <= '0';
23        END IF;
24        ----------Y(3):----------------------------
25        IF (temp1 >= temp2*8) THEN
26            y(3) <= '1';
27            temp1 := temp1-temp2*8;
28        ELSE y(3) <= '0';
29        END IF;
30        ----------Y(2):-----------------------------
31        IF (temp1 >= temp2*4) THEN
32            y(2) <= '1';
33            temp1 := temp1-temp2*4;
34        ELSE y(2) <= '0';
35        END IF;
36        ----------Y(1):-----------------------------
37        IF (temp1 >= temp2*2) THEN
38            y(1) <= '1';
39            temp1 := temp1-temp2*2;
40        ELSE y(1) <= '0';
41        END IF;
42        ----------Y(0):-----------------------------
43        IF (temp1 >= temp2) THEN
44            y(0) <= '1';
45            temp1 := temp1-temp2;
46        ELSE y(0) <= '0';
```

```
47          END IF;
48          ------Remainder: ----------------------------
49          rest <= temp1;
50      END PROCESS;
51 END rtl;
52 ------------------------------------------------
1  ----------方案 2: compact and generic-------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY divider IS
6      GENERIC(n: INTEGER := 3);
7      PORT (a, b: IN INTEGER RANGE 0 TO 15;
8            y: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
9            rest: OUT INTEGER RANGE 0 TO 15;
10           err: OUT STD_LOGIC);
11 END divider;
12 ------------------------------------------------
13 ARCHITECTURE rtl OF divider IS
14 BEGIN
15     PROCESS (a, b)
16        VARIABLE temp1: INTEGER RANGE 0 TO 15;
17        VARIABLE temp2: INTEGER RANGE 0 TO 15;
18     BEGIN
19        ------- Error and initialization: --------
20        temp1 := a;
21        temp2 := b;
22        IF (b=0) THEN err <= '1';
23        ELSE err <= '0';
24        END IF;
25        ---------------y: -----------------------
26        FOR i IN n DOWNTO 0 LOOP
27           IF (temp1 >= temp2*2**i) THEN
28              y(i) <= '1';
29              temp1 := temp1-temp2*2**I;
30           ELSE y(i) <= '0';
31           END IF;
32        END LOOP;
33        -----Remainder: -------------------------
```

```
34        rest <= temp1;
35     END PROCESS;
36 END rtl;
37 ---------------------------------------------------
```

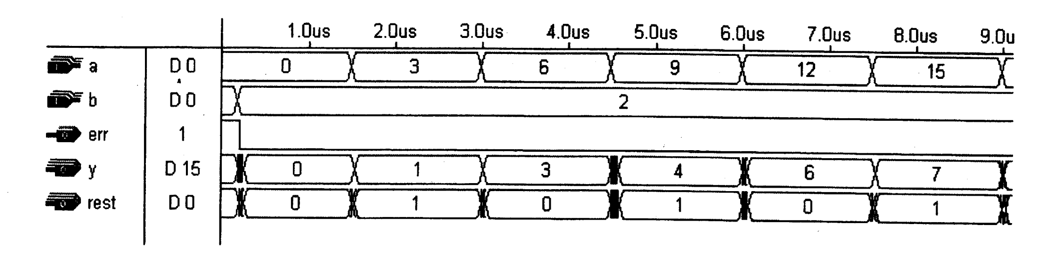


图 9.10 除法器的仿真结果

9.5 自动售货机控制器

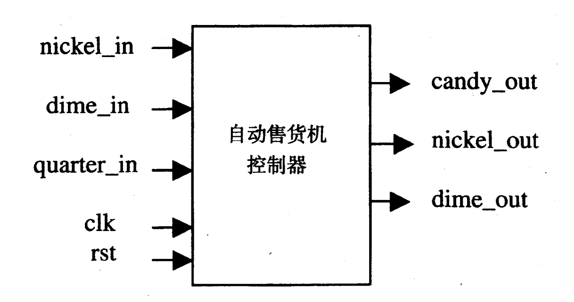
在这个例子中将设计一个自动售货机的控制器电路。该自动售货机销售价格为 25 美分的糖果。根据第 8 章所学的内容可以知道，这是一个利用有限状态机进行电路设计的典型例子。

控制器的输入和输出如图 9.11 所示。输入信号是 nickel_in（投入 5 美分），dime_in（投入 10 美分）和 quarter_in（存放 25 美分）。另外两个必要的输入是 clk（时钟）和 rst（复位）。控制器相应地有 3 个输出：candy_out 用于控制发放糖果，nickel_out 用于控制找回 5 美分的零钱，dime_out 用于控制找回 10 美分零钱。

图 9.11 给出了有限状态机的状态转移图。圆圈里的数代表顾客投进来的总钱数（接受 5 美分、10 美分或 25 美分的硬币）。状态 0 是空闲状态。从它开始，如果投入 5 美分硬币，将跳转到状态 5；如果投入 10 美分硬币，则跳转到状态 10；如果投入 25 美分硬币，则跳转到状态 25。随着投币数量的增加，状态不断跳转，如果投入的币值达到 25 美分，就可以进入状态 25，然后售货机会发放糖果，并跳转回状态 0。如果投入的总币值超过了 25 美分，那么售货机要进入与找零钱相关的状态。例如，当投入的币值达到 40 美分时，需要先退出 5 分硬币（进入状态 35），然后再退出 10 美分，发放糖果并进入状态 0。

整个设计包含两个主要问题，第一个是实现自动售货机控制器的基本功能（参见图 9.11），第二个是加入其他的特色。第一部分将在本节中进行研究，第二部分留在习题中（见习题 9.3）。从安全方面考虑，加入一些其他功能是很有必要的，因为在处理现金交易时要保证双方（机器或顾客）的利益在交易过程中不受损害。

下面的 VHDL 代码只处理了图 9.11 中描述的基本功能，辅助功能的添加留在习题 9.3 中。下面将采用 8.2 节中的设计风格#1 来实现该电路。



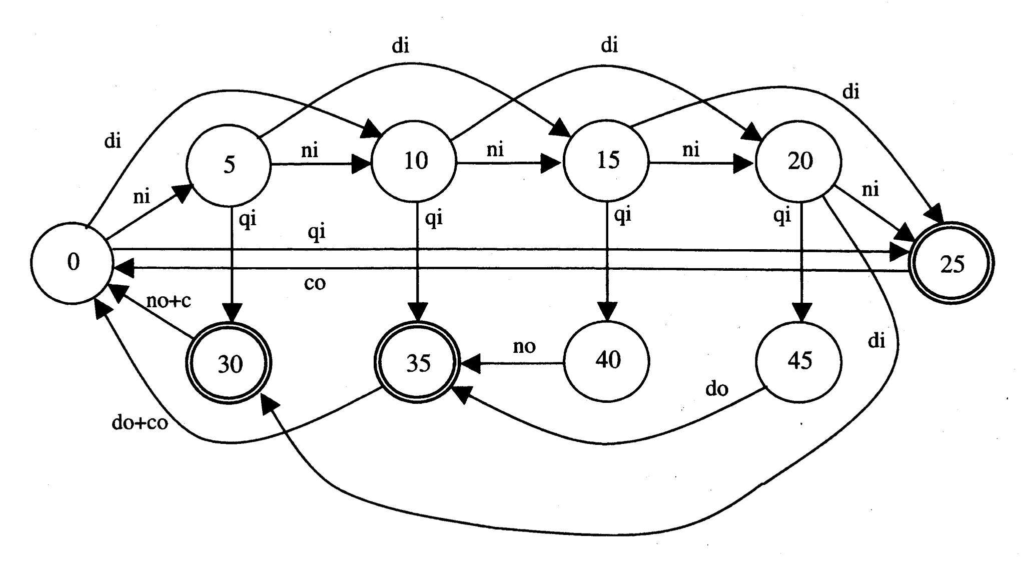


图 9.11　自动售货机控制器的顶层电路图和状态转移图。信号说明：ni = nickel_in，di = dime_in，qi = quarter_in，no = nickel_out，do = dime_out，co = candy_out

在代码的第 12 行中定义了枚举类型 state（参见图 9.11），它包含 10 个状态，所以至少需要用 4 位对其进行编码（将产生 4 个寄存器）。在默认状态下，编译器将按照它们的排列顺序对其进行编码，所以有 st0 = "0000"（十进制的 0），st5 = "0001"（十进制的 1），…和 st45 = "1001"（十进制的 9）。在仿真时，这些数字将代替状态的名称出现在仿真波形中。

```
1  --------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY vending_machine IS
6     PORT (clk, rst: IN STD_LOGIC;
7           nickel_in, dime_in, quarter_in: IN BOOLEAN;
```

```
8           candy_out, nickel_out, dime_out: OUT STD_LOGIC);
9  END vending_machine;
10 --------------------------------------------------
11 ARCHITECTURE fsm OF vending_machine IS
12    TYPE state IS (st0, st5, st10, st15, st20, st25,
13       st30, st35, st40, st45);
14    SIGNAL present_state, next_state: STATE;
15 BEGIN
16    ----Lower section of the FSM (Sec.8.2): ------
17    PROCESS (rst, clk)
18    BEGIN
19       IF (rst = '1') THEN
20          present_state <= st0;
21       ELSIF (clk'EVENT AND clk = '1') THEN
22          present_state <= next_state;
23       END IF;
24    END PROCESS;
25    ----Upper section of the FSM (Sec.8.2): -------
26    PROCESS (present_state, nickel_in, dime_in, quarter_in)
27    BEGIN
28       CASE present_state IS
29          WHEN st0 =>
30             candy_out <= '0';
31             nickel_out <= '0';
32             dime_out <= '0';
33             IF (nickel_in) THEN next_state <= st5;
34             ELSIF (dime_in) THEN next_state <= st10;
35             ELSIF (quarter_in) THEN next_state <= st25;
36             ELSE next_state <= st0;
37             END IF;
38          WHEN st5 =>
39             candy_out <= '0';
40             nickel_out <= '0';
41             dime_out <= '0';
42             IF (nickel_in) THEN next_state <= st10;
43             ELSIF (dime_in) THEN next_state <= st15;
44             ELSIF (quarter_in) THEN next_state <= st30;
45             ELSE next_state <= st5;
46             END IF;
```

```
47          WHEN st10 =>
48             candy_out <= '0';
49             nickel_out <= '0';
50             dime_out <= '0';
51             IF (nickel_in) THEN next_state <= st15;
52             ELSIF (dime_in) THEN next_state <= st20;
53             ELSIF (quarter_in) THEN next_state <= st35;
54             ELSE next_state <= st10;
55             END IF;
56          WHEN st15 =>
57             candy_out <= '0';
58             nickel_out <= '0';
59             dime_out <= '0';
60             IF (nickel_in) THEN next_state <= st20;
61             ELSIF (dime_in) THEN next_state <= st25;
62             ELSIF (quarter_in) THEN next_state <= st40;
63             ELSE next_state <= st15;
64             END IF;
65          WHEN st20 =>
66             candy_out <= '0';
67             nickel_out <= '0';
68             dime_out <= '0';
69             IF (nickel_in) THEN next_state <= st25;
70             ELSIF (dime_in) THEN next_state <= st30;
71             ELSIF (quarter_in) THEN next_state <= st45;
72             ELSE next_state <= st20;
73             END IF;
74          WHEN st25 =>
75             candy_out <= '1';
76             nickel_out <= '0';
77             dime_out <= '0';
78             next_state <= st0;
79          WHEN st30 =>
80             candy_out <= '1';
81             nickel_out <= '1';
82             dime_out <= '0';
83             next_state <= st0;
84          WHEN st35 =>
85             candy_out <= '1';
```

```
86              nickel_out <= '0';
87              dime_out <= '1';
88              next_state <= st0;
89          WHEN st40 =>
90              candy_out <= '0';
91              nickel_out <= '1';
92              dime_out <= '0';
93              next_state <= st35;
94          WHEN st45 =>
95              candy_out <= '0';
96              nickel_out <= '0';
97              dime_out <= '1';
98              next_state <= st35;
99  END CASE;
100 END PROCESS;
101
102 END fsm;
103 --------------------------------------------------
```

仿真的结果如图 9.12 所示。可以看到，整个仿真过程中一共投入了 3 个 5 美分硬币和 1 个 25 美分硬币。在第一个 5 美分硬币投进去后的第一个时钟上升沿出现时，有限状态机的状态从 st0（十进制的 0）转到了 st5（十进制的 1）。在投入第二个 5 美分硬币后状态转到 st10（十进制的 2），投入第三个 5 美分硬币后转到 st15（十进制的 3）。在 25 美分硬币投进去之后，状态变为 st40（十进制的 8）。此后，售货机退还顾客一个 5 美分硬币（nickel_out = '1'），同时状态机进入 st35（十进制的 7），接着退还 10 美分硬币（dime_out = '1'）并发放糖果（candy_out = '1'），同时系统回到空闲状态（st0）。

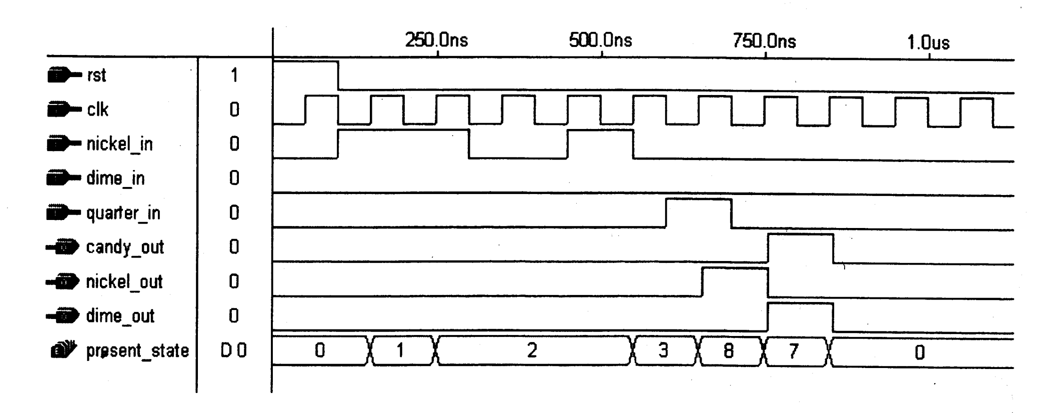


图 9.12　自动售货机电路的仿真波形

9.6　串行数据接收器

串行数据接收器电路如图 9.13 所示。它包括串行数据输入端口（din）和并行数据输出端口（data），输入端还包括时钟信号（6:0）。电路还产生了两个指示信号：err（错误指示信号）和 data_valid（有效数据指示信号）。

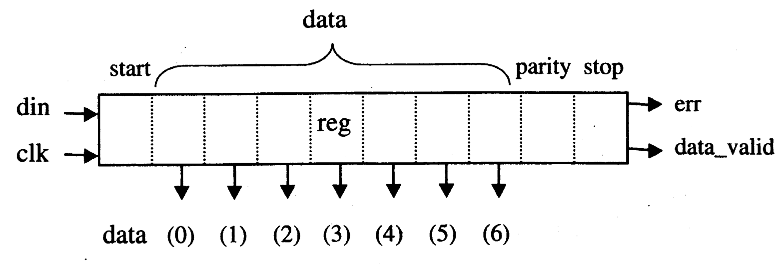


图 9.13　串行数据接收器

输入信号是一个字符流。每个完整的字符包括 10 位，第 1 位是起始位，当它为高时标志着一个字符的开始，电路应开始接收后面的数据。接下来的 7 位是有效数据，第 9 位是奇偶校验位，当数据中 1 的个数是偶数时其值是'0'，否则为'1'。第 10 位是终止位，如果传输正确，其值应该是'1'。当接收电路发现奇偶校验错误或者结束位不是'1'时将发出错误指示。当数据被正确接收时，存储在内部寄存器中的数据并行出现在 data（6:0）上，同时 data_valid 有效。

实现这个电路的 VHDL 代码如下。代码中使用了很多变量，如 count（计算接收的位数），reg（存储数据）和 temp（计算错误）等。在第 37 行中，用 reg(0) = din 来代替 reg(0) ='0'，这是因为在数据进行连续传输时，紧跟在终止位后面的时隙中传递的可能是下一个字符的起始位。

```
1  -------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY receiver IS
6     PORT (din, clk, rst: IN BIT;
7             data: OUT BIT_VECTOR(6 DOWNTO 0);
8             err, data_vaild: OUT BIT);
9  END receiver;
10 -------------------------------------------------
11 ARCHITECTURE rtl OF receiver IS
12 BEGIN
13    PROCESS (rst, clk)
14       VARIABLE count: INTEGER RANGE 0 TO 10;
```

```
15        VARIABLE reg: BIT_VECTOR (10 DOWNTO 0);
16        VARIABLE temp: BIT;
17     BEGIN
18        IF (rst = '1') THEN
19           count := 0;
20           reg := (reg'RANGE => '0');
21           temp := '0';
22           err <= '0';
23           data_vaild <= '0';
24        ELSIF (clk'EVENT AND clk = '1') THEN
25           IF (reg(0) = '0' AND din = '1') THEN
26              reg(0) := '1';
27           ELSIF (reg(0) = '1') THEN
28              count := count+1;
29              IF (count<10) THEN
30                 reg(count) := din;
31              ELSIF (count=10) THEN
32                 temp := (reg(1)XOR reg(2)XOR reg(3)XOR
33                          reg(4)XOR reg(5)XOR reg(6)XOR
34                          reg(7)XOR reg(8))OR NOT reg(9);
35                 err <= temp;
36                 count := 0;
37                 reg(0) := din;
38                 IF (temp = '0') THEN
39                    data_vaild <= '1';
40                    data <= reg(7 DOWNTO 1);
41                  END IF;
42              END IF;
43           END IF;
44        END IF;
45     END PROCESS;
46  END rtl;
47  -----------------------------------------------------
```

电路的仿真结果在图 9.14 中给出。输入的序列是{前导码 =1，数据 =0111001，奇偶校验位 =0，结束位 =1}。在上面一幅图中可以看到，这时没有检测到错误，因为奇偶校验位和结束位都是正确的。因此，在计数到 9 时，数据就有效了，也就是说，从 data(0)到 data(6)，数据 =0111001（十进制的 78），同时数据有效指示信号为高。在没有新的数据输入之前，输出数据始终保持为原来的值。

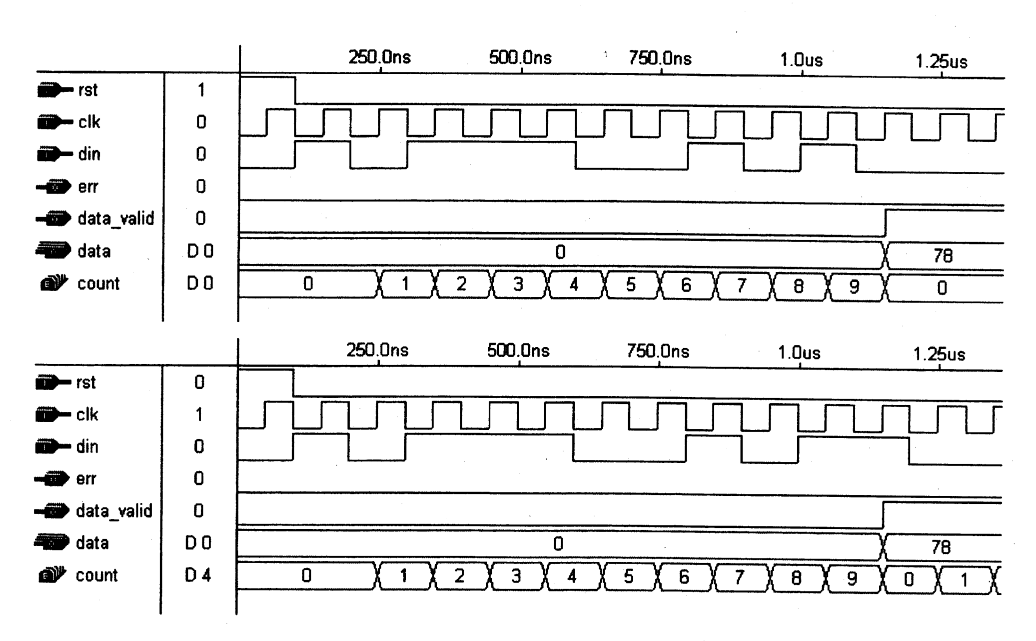


图 9.14　串行数据接收器的仿真结果

图 9.14 的下半部分与上半部分的惟一区别就是后面数据的前导码紧跟在前面数据的结束位后面。

9.7　并/串变换器

并/串变换器是一个使用移位寄存器的典型例子。它能够将并行的数据块以串行的方式连续输出。这种转换器的需求在不断增加，例如在 ASIC 芯片中，当没有足够的引脚来并行输出所有数据时就需要用这种转换器。

并/串转换器的电路结构如图 9.15 所示。d(7: 0)是需要发送的并行数据，dout 上是真正串行输出的数据。另外还有两个输入：clk 和 load。当 load 有效时，并行输入数据 d(7: 0)被同步存储在移位寄存器中。当 load 保持为高时，MSB，即 d(7)在输出端始终保持有效。一旦 load 返回'0'，接下来移位寄存器的各个位将在每个时钟上升沿依次出现在输出端口 dout 上。8 位数据全部发送完毕之后，输出端在下一次数据传输之前一直保持为低电平。

```
1      ------------------------------------------------
2      LIBRARY ieee;
3      USE ieee.std_logic_1164.all;
4      ------------------------------------------------
5      ENTITY serial_converter IS
6         PORT (d: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```
7              clk, load: IN STD_LOGIC;
8              dout: OUT STD_LOGIC);
9      END serial_converter;
10 -------------------------------------------------
11 ARCHITECTURE serial_converter OF serial_converter IS
12     SIGNAL reg: STD_LOGIC_VECTOR(7 DOWNTO 0);
13 BEGIN
14     PROCESS (clk)
15     BEGIN
16        IF (clk'EVENT AND clk = '1') THEN
17           IF (load = '1') THEN reg <= d;
18           ELSE reg <= reg(6 DOWNTO 0)& '0';
19           END IF;
20        END IF;
21     END PROCESS;
22     dout <= reg(7);
23 END serial_converter;
24 ---------------------------------------------------
```

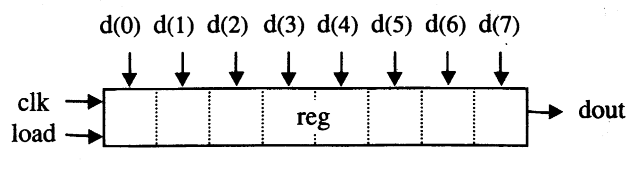


图 9.15　并/串转换器

用上面的代码综合出的电路的仿真结果如图 9.16 所示。这里的 d = "11011011"(十进制的 219)。可以看到，在 load 有效后的第一个时钟上升沿时，d(7)，出现在输出端口上，一直等到 load 变低 (为了证明这一点，我们将使 load 在两个周期的时间内保持高电平)。只要 load 返回'0'，其他的位就开始依次发送。当所有位都传输完毕后，输出变为低电平。

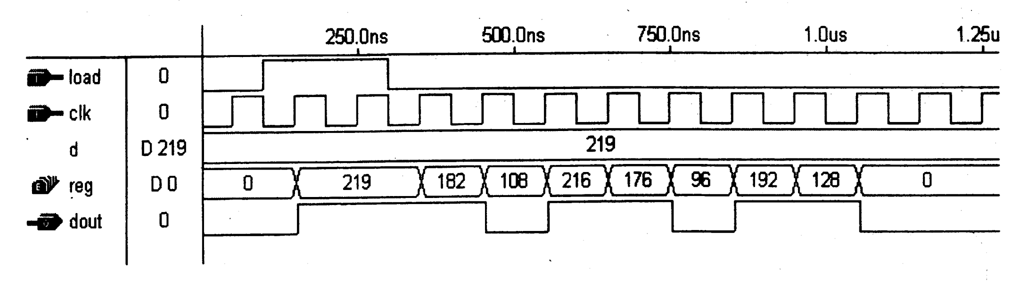


图 9.16　并/串转换器的仿真结果

9.8　一个 7 段显示器的应用例题

下面利用 SSD（seven-segment display，7 段显示器）来设计一个小游戏。电路的顶层结构如图 9.17 所示。它包括两个输入信号（clk 和 stop）和一个输出信号（dout）(6: 0)。输出信号连接到 SSD 上。这里的时钟频率 f_{clk} 为 1 kHz。

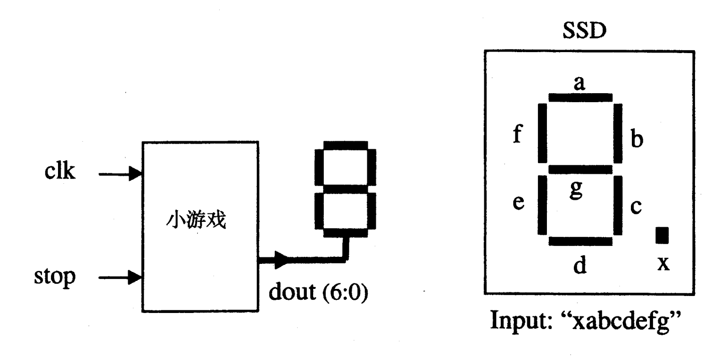


图 9.17　7 段显示器及驱动电路

我们所设计的电路要让 SSD 的各个段连续地按照顺时针方向转动。为了使循环运动的视觉效果更连贯，我们希望相邻两段在点亮的时间上略微重叠片刻。因此，点亮的次序应该是 a→ab→b→bc→c→cd→d→de→e→ef→f→fa→a。重叠的状态（ab，bc 和 cd 等）只维持几毫秒。如果 stop 为'1'，则电路返回状态 a，并在 stop 变低之前一直保持。

从第 8 章中可以清楚地知道，用 FSM 设计这个电路比较合适。图 9.18 给出了状态转移图。我们希望系统在 a，b 和 c 等状态时停留 80 ms（time1），在 ab，bc 和 cd 等状态时停留 30 ms（time2），然后才进入下一个状态。当电路的输入时钟周期为 1 ms 时，可以通过一个计数器的值得到已经等待了多少毫秒。在一个状态下，当计数器分别计数到 80 或 30 后才进入下一个状态。

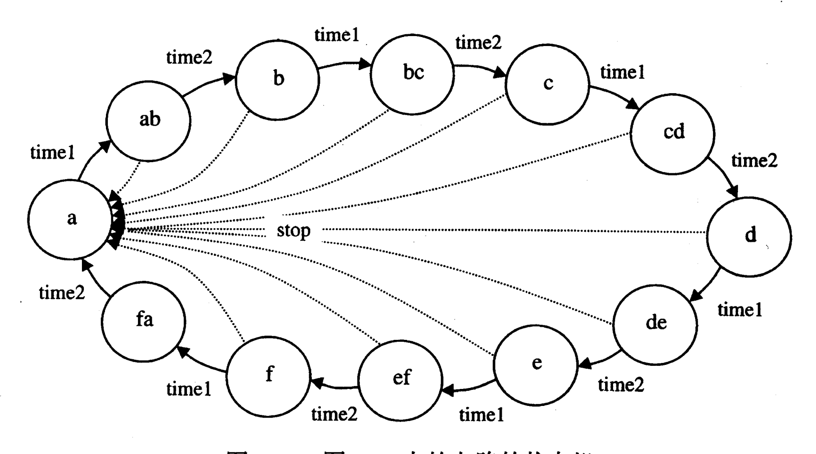


图 9.18　图 9.17 中的电路的状态机

下面给出了电路的 VHDL 代码。注意，这里直接使用了 8.2 节中的状态机设计模板。在第 11 行和第 12 行中，time1 和 time2 用两个常数声明。这里取值较小（分别为 4 和 2）是为了使仿真结果能在一张图中表现出来（实际应用时应分别取 80 和 30）。由于在不同状态下进行状态跳转时计数器的取值不同，这里使用了一个名为 flip 的信号来指出应该计数到 80 还是 30。

```
1  -------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY ssd_game2 IS
6     PORT (clk, stop: IN BIT;
7           dout: OUT BIT_VECTOR(6 DOWNTO 0));
8  END ssd_game2;
9  -------------------------------------------------
10 ARCHITECTURE fsm OF ssd_game2 IS
11    CONSTANT time1: INTEGER := 4; ---actual value is 80
12    CONSTANT time2: INTEGER := 2; ---actual value is 30
13    TYPE states IS(a, ab, b, bc, c, cd, d, de, e, ef, f, fa);
14    SIGNAL present_state, next_state: STATES;
15    SIGNAL count: INTEGER RANGE 0 TO 5;
16    SIGNAL flip: BIT;
17 BEGIN
18    -------Lower section of FSM(Sec.8.2): -------
19    PROCESS (clk, stop)
20    BEGIN
21       IF (stop = '1') THEN
22          present_state <= a;
23       ELSIF (clk'EVENT AND clk = '1') THEN
24          IF ((flip = '1' AND count=time1)OR
25             (flip = '0' AND count=time2)) THEN
26             count <= 0;
27             present_state <= next_state;
28          ELSE count <= count+1;
29          END IF;
30       END IF;
31    END PROCESS;
32    -------Upper section of FSM(Sec.8.2): -------
33    PROCESS (present_state)
34    BEGIN
```

```
35      CASE present_state IS
36        WHEN a =>
37          dout <= "1000000";  --Decimal 64
38          flip <= '1';
39          next_state <= ab;
40        WHEN ab =>
41          dout <= "1100000";  --Decimal 96
42          flip <= '0';
43          next_state <= b;
44        WHEN b =>
45          dout <= "0100000";  --Decimal 32
46          flip <= '1';
47          next_state <= bc;
48        WHEN bc =>
49          dout <= "0110000";  --Decimal 48
50          flip <= '0';
51          next_state <= c;
52        WHEN c =>
53          dout <= "0010000";  --Decimal 16
54          flip <= '1';
55          next_state <= cd;
56        WHEN cd =>
57          dout <= "0011000";  --Decimal 24
58          flip <= '0';
59          next_state <= d;
60        WHEN d =>
61          dout <= "0001000";  --Decimal 8
62          flip <= '1';
63          next_state <= de;
64        WHEN de =>
65          dout <= "0001100";  --Decimal 12
66          flip <= '0';
67          next_state <= e;
68        WHEN e =>
69          dout <= "0000100";  --Decimal 4
70          flip <= '1';
71          next_state <= ef;
72        WHEN ef =>
73          dout <= "0000110";  --Decimal 6
```

```
74              flip <= '0';
75              next_state <= f;
76           WHEN f =>
77              dout <= "0000010";  --Decimal 2
78              flip <= '1';
79              next_state <= fa;
80           WHEN fa =>
81              dout <= "1000010";  --Decimal 66
82              flip <= '0';
83              next_state <= a;
84         END CASE;
85     END PROCESS;
86 END fsm;
87 --------------------------------------------------
```

仿真结果如图 9.19 所示。可以看到，系统在 a，b 和 c 等状态停留了 4 个时钟周期（这里 time1 = 4），在 ab，bc 和 cd 等状态停留了两个时钟周期（这里 time2 = 2）。同样可以发现，仿真器仿真得到的十进制数与列举在代码中的十进制数完全吻合。

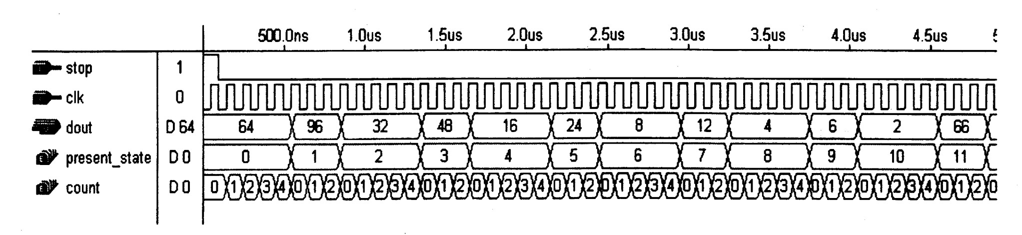


图 9.19　图 9.17 中的电路的仿真结果

9.9　信号发生器

根据输入的时钟信号，我们希望得到如图 9.20 所示的波形。要解决这样的问题，既可以使用有限状态机的方法，也可以使用传统的方法。下面将分别进行讨论。

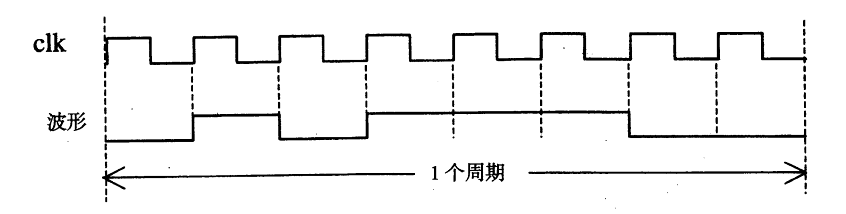


图 9.20　信号发生器需要产生的波形

采用 FSM 进行设计

根据图 9.20 的信号波形，可以设计一个具有 8 个状态的有限状态机。使用一个模 8 计数器的计数值代表有限状态机的 8 个状态。当 count = '0'时 wave = '0'（第一个脉冲），当 count = '1'时 wave = '1'，依次类推，就可以得到图 9.20 所示的波形。实现这个电路需要 4 个寄存器，其中 3 个用于计数器，另外 1 个用于存储 wave 值。从 8.2 节和 8.3 节可以知道，应该使用状态机设计风格#2，这时有限状态机的输出将被寄存。这一点非常重要，因为在信号发生器中不允许出现毛刺。

采用设计风格#2 编写的 VHDL 代码如下所示，仿真结果见图 9.21。检查由综合工具生成的报告文件，可以证实该电路实际使用了 4 个寄存器。

```
1  ---------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  ENTITY signal_gen IS
6     PORT (clk: IN STD_LOGIC;
7            wave: OUT STD_LOGIC);
8  END signal_gen;
9  ---------------------------------------------------
10 ARCHITECTURE fsm OF signal_gen IS
11    TYPE states IS (zero, one, two, three, four, five, six,
12                    seven);
13    SIGNAL present_state, next_state: STATES;
14    SIGNAL temp: STD_LOGIC;
15 BEGIN
16
17    -------Lower section of FSM (sec.8.3): ------
18    PROCESS (clk)
19    BEGIN
20       IF (clk'EVENT AND clk = '1') THEN
21          present_state <= next_state;
22          wave <= temp;
23       END IF;
24    END PROCESS;
25
26    ----Upper section of FSM (Sec.8.3):----------
27    PROCESS (present_state)
28    BEGIN
29       CASE present_state IS
```

```
30          WHEN zero => temp <= '0'; next_state <= one;
30          WHEN one => temp <= '1'; next_state <= two;
30          WHEN two => temp <= '0'; next_state <= three;
30          WHEN three => temp <= '1'; next_state <= four;
30          WHEN four => temp <= '1'; next_state <= five;
30          WHEN five => temp <= '1'; next_state <= six;
30          WHEN six => temp <= '0'; next_state <= seven;
30          WHEN seven => temp <= '0'; next_state <= zero;
38       END CASE;
39    END PROCESS;
40 END fsm;
41 -------------------------------------------------
```

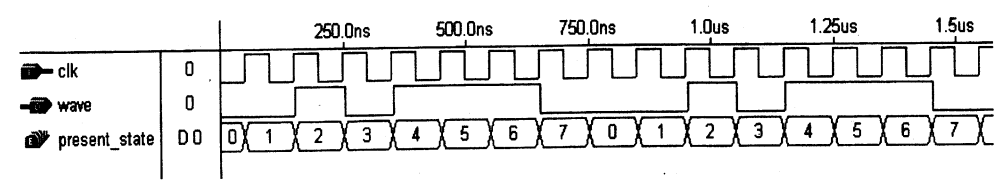


图 9.21 采用有限状态机的方法得到的仿真波形

采用传统方法进行设计

下面给出了使用 IF 语句的传统设计方法。由于计数器和输出波形都在时钟信号（clk）跳变时被赋值，因此根据在 7.5 节中所学的内容，两者都应该被存储起来（也就是说将占用 4 个寄存器，其中 3 个用于计数，另外 1 个用于波形输出）。仿真结果见图 9.22。

```
1  -------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY signal_gen1 IS
6     PORT (clk: IN BIT;
7           wave: OUT BIT);
8  END signal_gen1;
9  -------------------------------------------------
10 ARCHITECTURE arch1 OF signal_gen1 IS
11 BEGIN
12    PROCESS
13       VARIABLE count: INTEGER RANGE 0 TO 7;
14    BEGIN
15       WAIT UNTIL(clk'EVENT AND clk = '1');
16       CASE count IS
```

```
17            WHEN 0 => wave <= '0';
18            WHEN 1 => wave <= '1';
19            WHEN 2 => wave <= '0';
20            WHEN 3 => wave <= '1';
21            WHEN 4 => wave <= '1';
22            WHEN 5 => wave <= '1';
23            WHEN 6 => wave <= '0';
24            WHEN 7 => wave <= '0';
25         END CASE;
26         count := count+1;
27      END PROCESS;
28 ENDarch1;
29 ------------------------------------------------------------
```

图 9.22　采用传统设计方法得到的仿真波形

9.10　存储器设计

在这一节中将设计具有存储功能的以下电路：

- ROM
- 输入/输出数据总线分离的 RAM
- 具有双向 I/O 数据总线的 RAM

ROM（只读存储器）

图 9.23 给出了 ROM 的结构。因为它是只读存储器，所以不需要时钟信号和写使能信号。可以看出，电路内部包括预先存储的内容，并且由输入的地址（addr）来选择将哪个存储空间中的内容传送到输出端。对于这个设计，在一个寻址单元里可以存储一个 8 位的“字”。

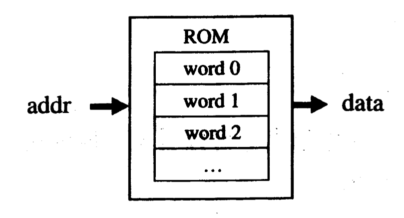


图 9.23　ROM 原型结构图

实现 ROM 的代码如下所示。words（第 7 行）表示存储在 ROM 中的字的数量，而 bits（第 6 行）表示每个字的大小。为了实现 ROM，使用了一个常数数组（第 15 行~第 22 行）。代码中首先定义了一个新的类型 vector_array（第 13 行~第 14 行），接着它被用来声明一个名为 memory（第 15 行）的常量。这个例子展示的是一个 8 × 8 的 ROM，下面的值（十进制）：0，2，4，8，16，32 和 128 分别被存储到地址 0~7 所选定的空间里（第 15 行~第 22 行）。第 24 行给出了使用 ROM 的例子。输出（data）等于 memory 存储在相应地址中的数据。当实现 ROM 时，由于没有使用时钟信号，所以没有占用寄存器资源。ROM 是通过使用逻辑门构成查找表（LUT：Look Up Table）的方式来实现的。

```
1  ----------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------------
5  ENTITY rom IS
6     GENERIC (bits: INTEGER := 8;       --# of bits per word
7              words: INTEGER := 8);   --# of words in the memory
8     PORT (addr: IN INTEGER RANGE 0 TO words-1;
9           data: OUT STD_LOGIC_VECTOR(bits-1 DOWNTO 0));
10 END rom;
11 ----------------------------------------------
12 ARCHITECTURE rom OF rom IS
13    TYPE vector_array IS ARRAY (0 TO words-1) OF
14       STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
15    CONSTANT memory: vector_array := ("00000000",
16                                      "00000010",
17                                      "00000100",
18                                      "00001000",
19                                      "00010000",
20                                      "00100000",
21                                      "01000000",
22                                      "10000000");
23 BEGIN
24    data <= memory(addr);
25 END rom;
26 ----------------------------------------------
```

ROM 的仿真工作波形在图 9.24 中给出。可以看见，地址从 0 到 7 连续改变，存储在相应位置上的数据可以相应地连续正确输出。

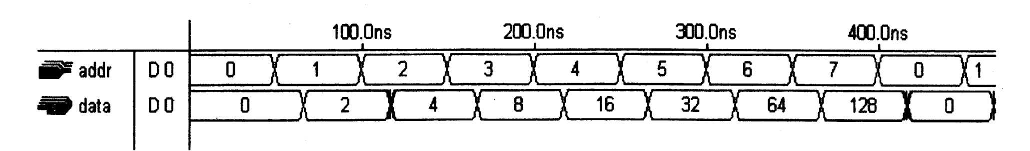


图 9.24　8 × 8 ROM 的仿真工作波形

输入/输出数据总线分离的 RAM

在图 9.25 中给出了具有独立输入/输出数据总线的 RAM（随机存储器）。实际上，这个电路已经在例 6.11 中讨论过了，在这里再次进行分析是为了与本节中的其他存储电路进行比较。

从图 9.25（a）中可以看到，该电路具有输入数据总线（data_in）、输出数据总线（data_out）、地址总线（addr）、时钟（clk）和写使能（wr_ena）引脚。如果写使能信号有效，在下一个时钟上升沿出现时，data_in 上的数据将被存储到地址总线所选择的位置上。另外，data_out 上始终显示当前地址总线所选择的存储单元中的内容。

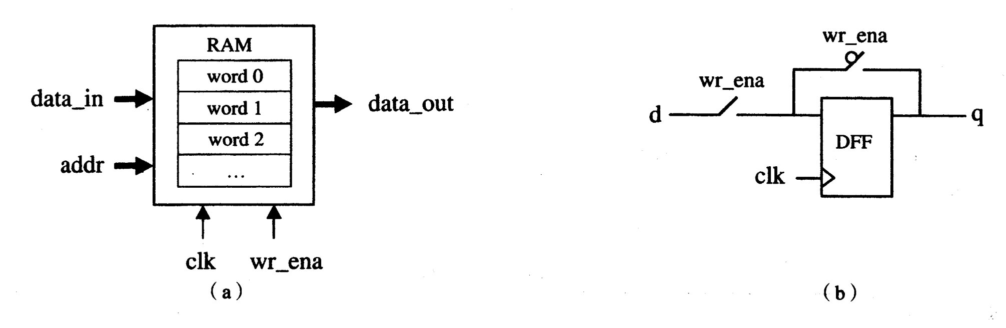


图 9.25　输入/输出数据总线分离的 RAM

从寄存器的观点来看，它的工作状况可以用图 9.25（b）进行辅助分析。当 wr_ena 为低时，q 连接到寄存器的输入端，外部输入 d 是断开的，所以不会有新的数据被写入寄存器中。当 wr_ena 变为高时，d 连接到寄存器的输入端，当时钟上升沿到达时，将存储新的输入数据。

实现图 9.25 中的电路的 VHDL 代码如下所示。它的容量是 16 字节。为了增加通用性，与存储容量相关的参数都在 GENERIC 中进行了定义。电路的仿真工作波形如图 9.26 所示。

```
1  --------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY ram IS
6     GENERIC(bits: INTEGER := 8;          -- #每字的位宽
7             words: INTEGER := 16);       -- #存储器中的
8                                          -- 字数
9     PORT (wr_ena, clk: IN STD_LOGIC;
```

```
10          addr: IN INTEGER RANGE 0 TO words-1;
11          data_in: IN STD_LOGIC_VECTOR(bits-1 DOWNTO 0);
12          data_out: OUT STD_LOGIC_VECTOR(bits-1 DOWNTO 0));
13 END ram;
14 -------------------------------------------------
15 ARCHITECTURE ram OF ram IS
16    TYPE vector_array IS ARRAY (0 TO words-1) OF
17       STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
18    SIGNAL memory: vector_array;
19 BEGIN
20    PROCESS (clk, wr_ena)
21    BEGIN
22       IF (wr_ena = '1') THEN
23          IF (clk'EVENT AND clk = '1') THEN
24             memory(addr) <= data_in;
25          END IF;
26       END IF;
27    END PROCESS;
28    data_out <= memory(addr);
29 END ram;
30 -------------------------------------------------
```

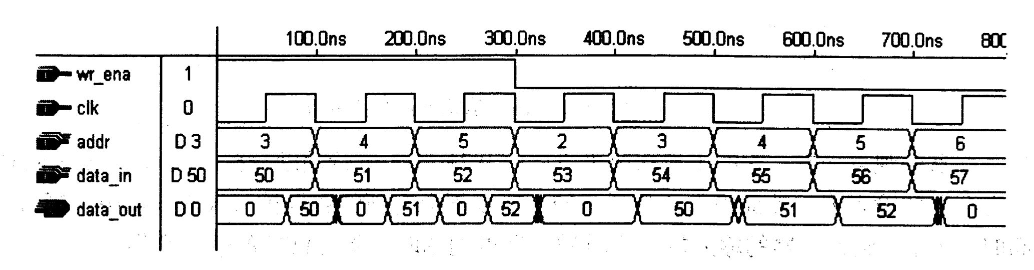


图 9.26 输入/输出数据总线分离的 16×8 RAM 的仿真工作波形

具有双向 I/O 数据总线的 RAM

图 9.27 中给出的 RAM 的数据总线是双向的。总体结构和图 9.25 相似，不同的是这里的数据总线是双向的。

从寄存器的角度来看，这个电路的工作原理可以用图 9.27（b）所示的电路进行类比分析。当 wr_ena 为低时，寄存器的输出与它的输入相连，同时总线处于数据输出状态，相应地址选择的数据被输出到总线上。当 wr_ena 有效时，数据总线上的数据直接与 d 连接，在下一个时钟周期的上升沿到达时，数据被写入到寄存器中。

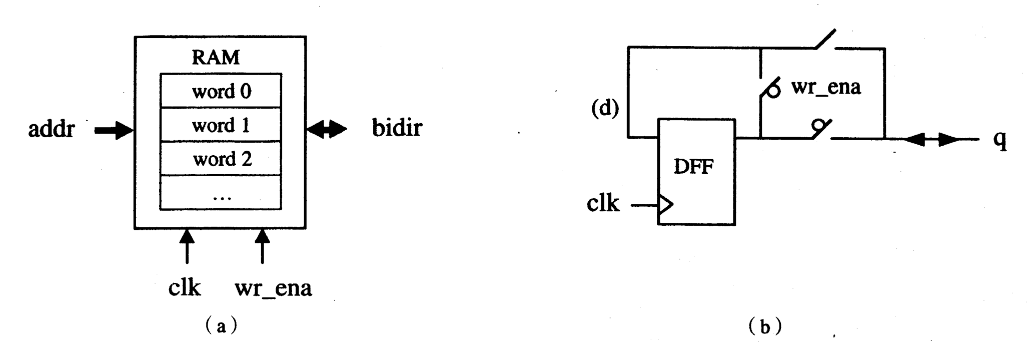


图 9.27　具有双向数据总线的 RAM

实现图 9.27 所示电路的 VHDL 代码如下。RAM 的容量是 16 字节。同样，与 RAM 容量相关的参数都在 GENERIC 中进行了定义。仿真结果在图 9.28 中给出。

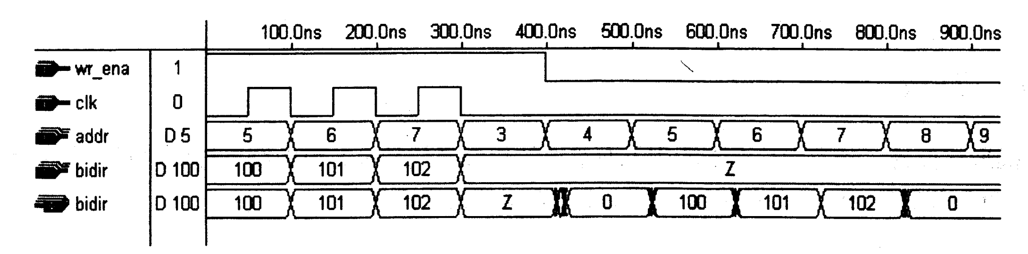


图 9.28　具有双向 I/O 数据总线的 16×8 RAM 的仿真工作波形

```
1  ----------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------------------
5  ENTITY ram4 IS
6     GENERIC(bits: INTEGER := 8;         -- 每字的位宽
7              words: INTEGER := 16);      -- 存储器中的
8                                          -- 字数
9     PORT (clk, wr_ena: IN STD_LOGIC;
10          addr: IN INTEGER RANGE 0 TO words-1;
11          bidir: INOUT STD_LOGIC_VECTOR(bits-1 DOWNTO 0);
12 END ram4;
13 ---------------------------------------------------
14 ARCHITECTURE ram OF ram4 IS
15    TYPE vector_array IS ARRAY (0 TO words-1) OF
16       STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
17    SIGNAL memory: vector_array;
```

```
18 BEGIN
19    PROCESS (clk, wr_ena)
20    BEGIN
21       IF (wr_ena = '0') THEN
22          bidir <= memory(addr);
23       ELSE
24          bidir <= (OTHERS => 'z');
25          IF (clk'EVENT AND clk = '1') THEN
26             memory(addr) <= bidir;
27          END IF;
28       END IF;
29    END PROCESS;
30 END ram;
31 --------------------------------------------------
```

9.11 习题

9.1 **桶形移位器**。分析为什么不能用下面给出的代码来代替 9.1 节中给出的桶形移位寄存器中的 ARCHITECTURE？两者相比，哪种描述方式的代码更简短？

```
-----------------------------------------------------
ARCHITECTURE barrel OF barrel IS
BEGIN
   PROCESS (inp, shift)
   BEGIN
      IF (shift = 0) THEN
         outp <= inp;
      ELSE
         FOR i IN 0 TO shift-1 LOOP
            outp(i) <= '0';
         END LOOP;
         FOR i IN shift TO inp'HIGH LOOP
            outp(i) <= inp(i-1);
         END LOOP;
      END IF;
   END PROCESS;
END barrel;
```

9.2　除法器。在 9.4 节中，我们研究了定点除法器的设计。当时介绍的两种实现方法都使用了顺序描述（IF 和 LOOP）语句，此外还编写了该节所给出的第二种除法算法的代码。现在这个习题要求采用并发描述语句实现除法运算（使用 GENERATE），除法算法如图 9.9 所示。为了做到这一点，建议构造和使用下面的类型和信号：

```
SUBTYPE long IS STD_LOGIC_VECTOR (2n DOWNTO 0);
TYPE vec_array IS ARRAY(n DOWNTO 0) OF long;
SIGNAL a_input, b_input: vec_array;
```

其中 n 的具体数值由 GENERIC 参数确定。

9.3　自动售货机控制器。下面要对 9.5 节中设计的自动售货机控制器在以下几个方面进行改进。

（a）为了提供必要的安全性能，需要在控制器和外围电路之间加入一些握手信号。握手信号可以包括下面的内容：

（i）输入有效指示信号（coin_valid）：外围电路使用该信号通知控制器新的输入可以被读取了。这个信号一旦被控制器处理，就返回'0'，所以只有在有效指示信号出现上升沿时才考虑读取新的输入。这种做法可以避免在 nickel_in，dime_in 和 quarter_in 保持有效的时间大于一个时钟周期时出现对投入钱币次数上的错误判断。

（ii）输入确认信号（称为 coin_accept）：它从控制器传给外围电路，表示刚才的输入已被处理了。一旦收到这个信号，外围电路就会将 coin_valid 置为'0'。

（b）考虑到机器中的 5 美分或 10 美分的硬币盒可能会变空，在状态机中设计零钱找兑的返回路径时要考虑这种情况（建议在图 9.11 所示的状态机中加入 st45→st40 和 st35→st30 的状态转换路径）。

（c）最后还要考虑顾客所投入的币值已经达到或超过 25 美分后仍然继续投币的情况，试分析此时应如何处理？

9.4　串行数据接收器。试用有限状态机的方法设计 9.6 节的串行数据接收器（见第 8 章）。在开始写 VHDL 代码之前，先将系统的状态转移图画出来。

9.5　串行数据发送器。这与 9.6 节中处理的问题正好相反。这里要实现的是将并行输入的数据串行输出。具体的电路结构如图 P9.5 所示。其数据组织的方式和 9.6 节中的相同。也就是说，包括 1 位前导码、7 位用户数据、奇偶校验位和 1 位结束位。图中的 data_ready 用于指示当前的输入数据是否可以被加载到寄存器中进行串行发送。

9.6　7 段数码显示器的应用。在 9.8 节的 7 段数码显示驱动电路中加入以下新特色：

（a）增加一个 2 位的输入信号，名称为 speed。它可以选择 4 种不同循环运动速度。重叠时间（time2）始终保持为 30 ms，仅改变 time1。实现这个电路并在 4 种不同循环周期上验证设计的正确性。

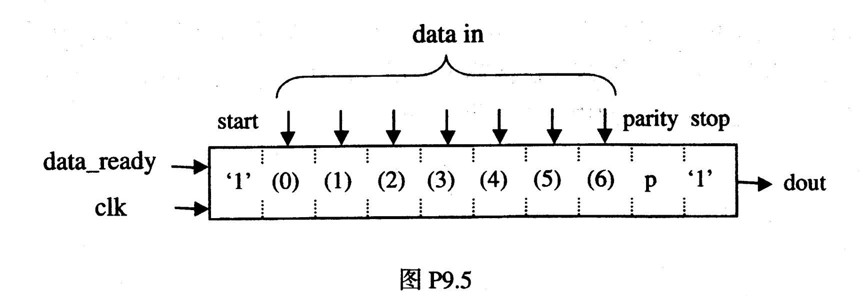


图 P9.5

(b) 改变输入端口 stop 的功能，例如当结束有效时不是跳转到状态 a，而是固定在 stop 生效时所处的状态。当 stop 又跳变为'0'时继续循环。

(c) 最后加入一个 direction 引脚。当其为低时，电路的工作方式与上面相同，当其为'1'时，SSD 的各段按照逆时针方向转动。

(d) 电路的物理实现：在上面的设计完成综合和仿真之后，将最终生成的网表加载到 PLD/FPGA 上，再按照下面的步骤进行测试：

(i) 首先，根据编译器生成的报告文件，确认输入引脚（clk，switch）和输出引脚（dout）的分配是正确的。

(ii) 接下来将信号发生器的输出信号频率设定为 1 kHz，并选择合适的输出电平，然后将其与 clk 引脚相连。此后检查电路供电是否正确。

(iii) 将电路的输出连接到 SSD 上。

(iv) 最后将 CPLD/FPGA 开发环境生成的网表下载到开发板上，并输入时钟信号，观察电路的工作情况，注意应对所有工作模式都进行测试。

9.7 速度监控器。图 P9.7 给出了一种汽车速度监控器。它具有如下工作特点：

- 每按一次速度选择按钮（SPEED），选择下一个被监视的速度（35，45，55，60，65，70，75 或 80 mil/h）。
- 针对每种被监视的速度，设有一个 LED，选中一个速度后，对应的 LED 就会被点亮。
- 两个 SSD 是用来显示汽车的当前速度的。汽车的速度计提供一个时钟信号，它的时钟频率与速度成比例。可以查看所选用速度计的数据手册或者先随意选取一个参数（如速度每增加 1 mil，频率增加 100 Hz）来对电路进行测试。
- 当汽车接近现在设定的监视速度时应使用蜂鸣器发出预警声。当速度离设定值还差 3 mil 或更少时应发出频率为 2 Hz 的断续蜂鸣信号。当速度超出设定值时，应发出连续的警报信号。这里可以考虑使用一个带有内部振荡器的蜂鸣器，此时外部只要提供直流驱动信号就可以使其发出连续警报声。在预警的情况下，驱动信号是 2 Hz 的方波。

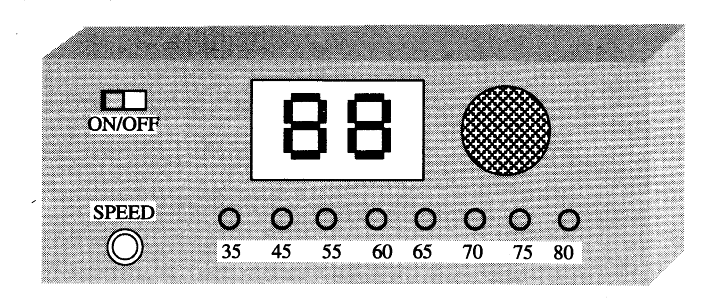


图 P9.7

9.8　随机数据发生器。设计一个随机数据发生器。数据取值范围是从"0000"（显示 0）到"1111"（显示 F）。完成对代码的编译和综合后，在 PLD/FPGA 开发工具上进行实际验证。

第二部分　系统设计

第 10 章　包集和元件

10.1　概述

本书第一部分研究了 VHDL 的背景知识和代码编写技巧，包括以下内容：

- 代码结构：库的声明、实体、构造体（第 2 章）
- 数据类型（第 3 章）
- 操作符及属性（第 4 章）
- 并发描述与并发代码（第 5 章）
- 顺序描述与顺序代码（第 6 章）
- 信号、变量和常量（第 7 章）
- 有限状态机的设计（第 8 章）
- 电路设计例题分析（第 9 章）

根据图 10.1 可以知道，我们已经学习了 VHDL 代码的主要构成部分，无论设计小电路或大系统，都必须很好地理解这些内容。

本书第二部分将在以上内容的基础上学习以下内容（如图 10.1 中右图所示）：

- 包集（第 10 章）
- 元件（第 10 章）
- 函数（第 11 章）
- 过程（第 11 章）

这些新的组成部分被添加到代码主体部分的目的是实现常用代码的共享（见图 10.1）。这些常用代码一般都被放在库文件（LIBRARY）中。设计者可以将自己设计的一些常用代码段添加到 LIBRARY 中，这有利于使一个复杂设计具有更清晰的结构。总之，经常使用的代码可以以元件（COMPONENT）、函数（FUNCTION）或过程（PROCEDURE）等形式放到 PACKAGE 中，然后被编译到目标 LIBRARY 中。

通过第 2 章的学习已经知道，在设计中经常使用的 3 个库是 ieee，std 和 work。在学习了后面的章节后，将能够建立自己的库，并像这 3 个库一样方便地使用。

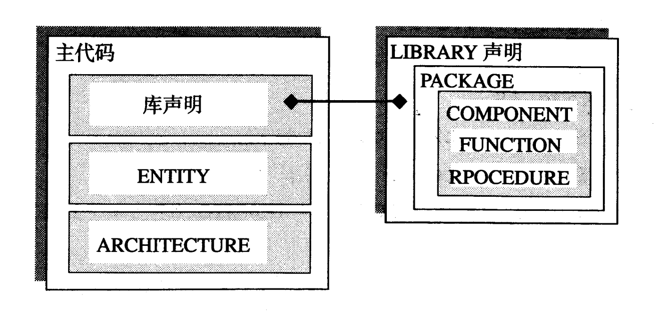


图 10.1　VHDL 代码的基本构成

10.2　包集

如上所述，经常使用的 VHDL 代码段通常以 COMPONENT，FUNCTION 或 PROCEDURE 的形式编写。这些代码被添加到 PACKAGE 中，并在最后编译到目标 LIBRARY 中。使用这种方法非常重要的原因是它允许代码分割、代码共享和代码重用。

除了 COMPONENT，FUNCTION 和 PROCEDURE 之外，PACKAGE 中还包括类型（TYPE）和常量（CONSTANT）的定义。

其语法格式如下：

```
PACKAGE package_name IS
    (声明)
END package_name;
[PACKAGE BODY package_name IS
    (FUNCTION 和 PROCEDURE 描述)
END package_name; ]
```

可以看出，上面的语法结构中包括两部分：PACKAGE 和 PACKAGE BODY。第一部分是必需的，包括所有声明语句。如果在第一部分中有一个或多个 FUNCTION 或 PROCEDURE 声明，那么在 PACKAGE BODY 中一定要存在对应的描述代码。PACKAGE 和对应的 PACKAGE BODY 的名称必须相同。

PACKAGE 中的声明部分包括以下内容：元件、函数、过程、类型和常量说明等。

例 10.1　简单的程序包

下面的例子中给出了一个名为 my_package 的 PACKAGE。它仅包括类型和常量的声明，所以不需要 PACKAGE BODY。

```
1 --------------------------------------------------
2 LIBRARY ieee;
```

```
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  PACKAGE my_package IS
6     TYPE state IS (st1, st2, st3, st4);
7     TYPE color IS (red, green, blue);
8     CONSTANT vec: STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111111";
9  END my_package;
10 -------------------------------------------------
```

例 10.2 内部包含函数的 PACKAGE

这个例子除了对类型和常量进行说明以外，还包括一个函数。因此使用需要 PACKAGE BODY。这个函数在时钟上升沿出现时返回 TURE。

```
1  -------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  PACKAGE my_package IS
6     TYPE state IS (st1, st2, st3, st4);
7     TYPE color IS (red, green, blue);
8     CONSTANT vec: STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111111";
9     FUNCTION positive_edge(SIGNAL s: STD_LOGIC)RETURN BOOLEAN;
10 END my_package;
11 -------------------------------------------------
12 PACKAGE BODY my_package IS
13    FUNCTION positive_edge(SIGNAL s: STD_LOGIC)RETURN BOOLEAN IS
14    BEGIN
15       RETURN(s'EVENT AND clk = '1');
16    END positive_edge;
17 END my_package;
18 -------------------------------------------------
```

上面两个例题中的 PACKAGE 都可以被编译成 work 或其他任何一个库中的一部分。为了在 VHDL 代码中使用它，必须在主程序中加入一个新的 USE 子句，具体方法如下所示：

```
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_package.all;
-------------------------------------------------
```

```
ENTITY...
...
ARCHITECTURE...
...
------------------------------------------------
```

10.3　元件

一个元件（COMPONENT）是一段结构完整的常用代码（包括库声明、实体和结构体这些基本组成部分，见第 2 章）。然而，如果将这些代码声明为 COMPONENT，就可以被其他电路调用，从而使代码具有了层次化的结构。

元件是一种进行代码分割、代码共享和代码重用的方法。例如，常用的触发器、乘法器、加法器和基本门电路等都可以放到一个 LIBRARY 中，供所有设计者方便地进行调用。

为了使用（实例化）一个元件，首先要对这个元件进行声明。其相应的格式如下：

元件声明：

```
COMPONENT component_name IS
    PORT (
      port_name: signal_mode signal_type;
      port_name: signal_mode signal_type;
      ...);
END COMPONENT;
```

元件实例化：

```
label: component_name PORT MAP (port_list);
```

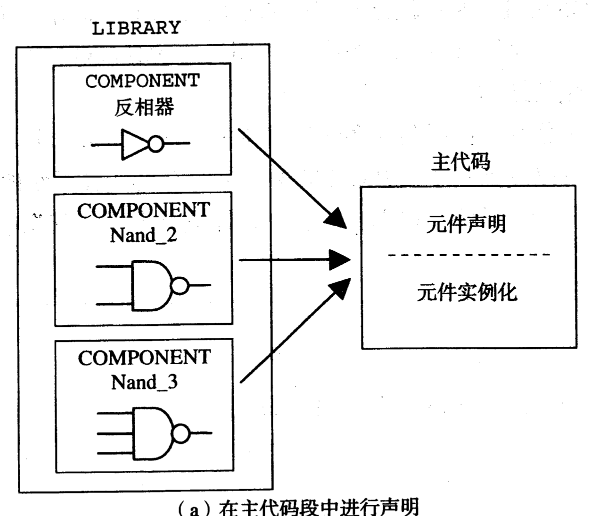
可以看出，COMPONENT 和 ENTITY 的声明格式一样（见 2.3 节），必须指出端口名称以及端口的模式（IN，OUT，BUFFER 或 INOUT）以及数据类型（STD_LOGIC_VECTOR，INTEGER 和 BOOLEAN 等）。为了实例化元件，在元件名和端口映射声明的前面还需要加入一个标号。端口列表将元件预定义的端口和实例化时的实际端口关联起来。

例　我们以一个反相器为例进行说明，该反相器已经设计完成（inverter.vhd）并编译到 work 库中。通过下面的代码来调用它。这个元件所选用的标号是 U1。实际电路中的端口名是 x 和 y，分别与 work 库中反相器的 a 和 b 相连（这种方式被称为位置映射，两者端口的排列顺序必须一一对应）。

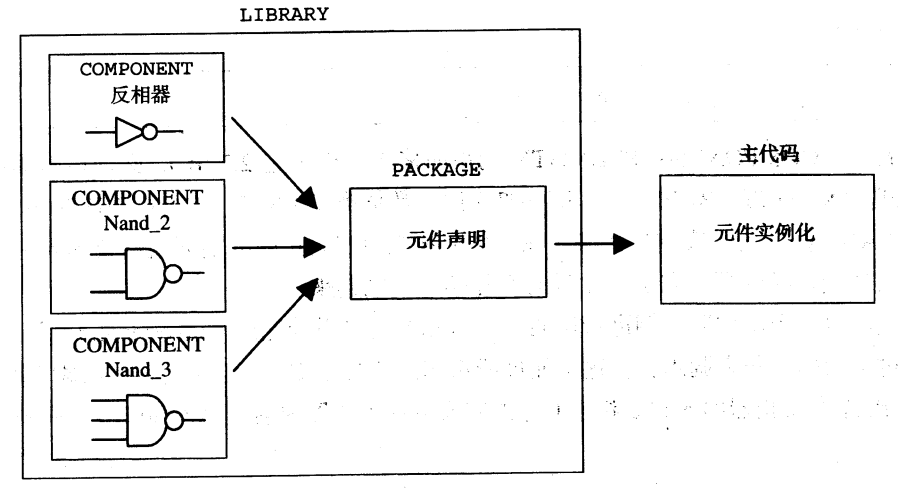
```
---------------------元件声明：-------------------
COMPONENT inverter IS
      PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
```

```
END COMPONENT;
--------------------元件实例化: --------------------
U1: inverter PORT MAP (x, y);
```

声明一个元件有两种基本方法（见图10.2）。一旦将元件设计完成并放到目标库中，就可以采用如图 10.2（a）所示的方式在主代码中进行声明。还可以采用如图 10.2（b）所示的方式，使用包集的方式进行声明。后者可以避免每次元件实例化时都要重复声明。下面分别介绍运用这两种不同方法的例子。



（a）在主代码段中进行声明



（b）在 PACKAGE 中进行声明

图 10.2　元件声明的基本方法

例 10.3　在主代码中声明元件

下面希望通过元件声明的第一种方式来实现图 10.3 所示的电路。首先需要设计每个元件的 VHDL 代码，然后设计主代码。由于没有将这些元件加入到 PACKAGE 中，所以元件必须在主代码中声明。仿真结果见图 10.4。

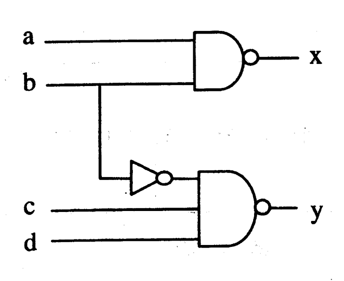


图 10.3　例 10.3 中的电路

```
1  -------------File inverter.vhd: -----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY inverter IS
6     PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
7  END inverter;
8  -------------------------------------------------
9  ARCHITECTURE inverter OF inverter IS
10 BEGIN
11    b <= NOT a;
12 END inverter;
13 -------------------------------------------------
```

```
1  ---------------File nand_2.vhd: -----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY nand_2 IS
6     PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
7  END nand_2;
8  -------------------------------------------------
9  ARCHITECTURE nand_2 OF nand_2 IS
10 BEGIN
11    c <= NOT (a AND b);
```

```
12 END nand_2;
13 ---------------------------------------------

1  ----------------File nand_3.vhd: ----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY nand_3 IS
6     PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
7  END nand_3;
8  ---------------------------------------------
9  ARCHITECTURE nand_3 OF nand_3 IS
10 BEGIN
11    d <= NOT (a AND b AND c);
12 END nand_3;
13 ---------------------------------------------

1  --------------File project.vhd: -----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY project IS
6     PORT (a, b, c, d: IN STD_LOGIC;
7           x, y: OUT STD_LOGIC);
8  END project;
9  ---------------------------------------------
10 ARCHITECTURE structural OF project IS
11    ---------------------------------------------
12    COMPONENT inverter IS
13       PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
14    END COMPONENT;
15    ---------------------------------------------
16    COMPONENT nand_2 IS
17       PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
18    END COMPONENT;
19    ---------------------------------------------
20    COMPONENT nand_3 IS
21       PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
22    END COMPONENT;
```

```
23   ----------------------------------------------
24     SIGNAL w: STD_LOGIC;
25 BEGIN
26     U1: inverter PORT MAP(b, w);
27     U2: nand_2    PORT MAP(a, b, x);
28     U3: nand_3    PORT MAP(w, c, d, y);
29 END structural;
30 ------------------------------------------------
```

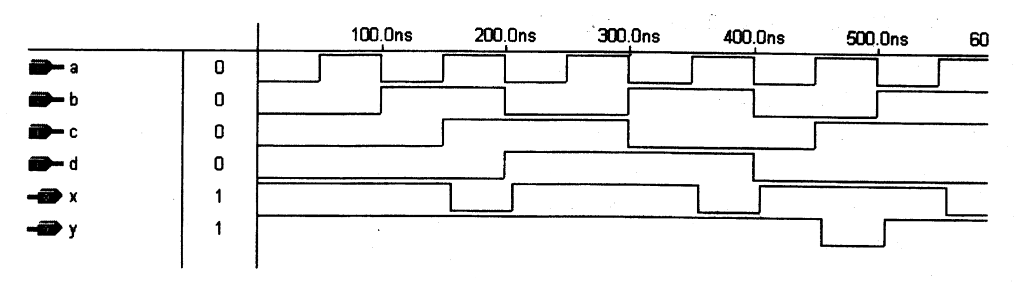


图 10.4　例 10.3 的仿真结果

例 10.4　在包集中声明元件

下面利用第二种元件声明的方法实现与前一例题相同的电路。如图 10.2（b）所示，我们需要创建一个包集，在包集中对所有元件进行声明。与例 10.3 相比，这里需要增加一段创建自己的 PACKAGE 的代码。这里虽然多了一个文件，但这个文件只需要创建一次，关键是可以避免在主代码中每实例化一个元件就声明一次。

这里需要注意的是，在主代码中应该增加一条 USE 语句，使新创建的包集 my_components 可以用于设计中。下面的代码的仿真波形显然与例 10.3 的是相同的。

```
1  ---------------File inverter.vhd: ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY inverter IS
6      PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
7  END inverter;
8  ------------------------------------------------
9  ARCHITECTURE inverter OF inverter IS
10 BEGIN
11    b <= NOT a;
12 END inverter;
13 ------------------------------------------------
```

```
1  ----------------File nand_2.vhd: ----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY nand_2 IS
6     PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
7  END nand_2;
8  ------------------------------------------------
9  ARCHITECTURE nand_2 OF nand_2 IS
10 BEGIN
11    c <= NOT (a AND b);
12 END nand_2;
13 ------------------------------------------------
1  ---------------File nand_3.vhd: -----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY nand_3 IS
6     PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
7  END nand_3;
8  ------------------------------------------------
9  ARCHITECTURE nand_3 OF nand_3 IS
10 BEGIN
11    d <= NOT (a AND b AND c);
12 END nand_3;
13 ------------------------------------------------

1  ------------File my_components.vhd: -------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  PACKAGE my_components IS
6     ------------------inverter: -----------------
7     COMPONENT inverter IS
8        PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
9     END COMPONENT;
10    ---------------2-input nand: ----------------
11    COMPONENT nand_2 IS
12       PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
```

```
13    END COMPONENT;
14    ----------------3-input nand: ----------------
15    COMPONENT nand_3 IS
16       PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
17    END COMPONENT;
18    ------------------------------------------------
19 END my_components;
20 --------------------------------------------------
1  --------------File project.vhd: -----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_components.all;
5  --------------------------------------------------
6  ENTITY project IS
7     PORT (a, b, c, d: IN STD_LOGIC;
8           x, y: OUT STD_LOGIC);
9  END project;
10 --------------------------------------------------
11 ARCHITECTURE structural OF project IS
12    SIGNAL w: STD_LOGIC;
13 BEGIN
14    U1: inverter PORT MAP(b, w);
15    U2: nand_2   PORT MAP(a, b, x);
16    U3: nand_3   PORT MAP(w, c, d, y);
17 END structural;
18   ------------------------------------------------
```

10.4　端口映射

在元件实例化过程中，有两种方法可以用来实现元件端口的映射：位置映射和名称映射。我们来看看下面的例子：

```
COMPONENT inverter IS
   PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
END COMPONENT;
...
U1: inverter PORT MAP(x, y);
```

例子中采用的是位置映射的方法，端口 x 和 y 对应于 a 和 b。下面的例子采用的是名称映射的方法：

```
U1: inverter PORT MAP(x => a, y => b);
```

一般说来，位置映射书写较为简单，但名称映射的方法不容易出错。对于不需要使用的端口可以断开，但这时需要使用关键词 OPEN。比如：

```
U2: my_circuit PORT MAP(x => a, y => b, w => OPEN, z => d);
```

10.5 GENERIC 参数的映射

如果元件实例化时需要通过 GENERIC 传递参数（见 4.5 节），那么一定需要进行 GENERIC 参数的映射。此时，元件实例化的格式需要进行如下更新：

```
lable: compon_name GENERIC MAP(param.list)PORT MAP(port list);
```

可以看到，它增加了 GENERIC 参数映射部分，目的是进行 GENERIC 参数的说明和传递。下面将举例说明 GENERIC 映射的应用。

例 10.5 带有 GENERIC 参数的元件的实例化

下面分析与例 4.3 相同的通用奇偶校验产生器（见图 10.5），当输入矢量中'1'的个数为偶数时，在输入矢量中插入一个'0'，当输入矢量中'1'的个数为奇数时则插入一个'1'，从而使输出矢量中始终包含偶数个'1'。

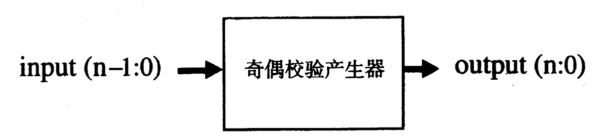


图 10.5 例 10.5 的通用奇偶校验产生器电路

下面的代码的通用性非常强，因为输入矢量的宽度 n 是 Generic 参数，也就是说，n 可以是任何正整数。下面包括两个文件，一个与元件有关（实际上可以假设 par_generator 是以前就已经设计完成并加入到 work 库中的），另一个与主代码本身有关，此处需要实例化元件 par_generator。

这里需要注意的是，元件设计文件（parity_gen）中的默认矢量宽度（n = 7）将在元件实例化过程中被 GENERIC 映射时提供的新参数（n = 2）覆盖。另外，在第二个文件中进行元件声明时，必须进行 GENERIC 声明，因为它是元件 ENTITY 的一个组成部分。然而，这里没有必要再次声明它的默认值。下面的代码综合后的仿真结果如图 10.6 所示。

```
1  ---------File parity_gen.vhd (component): -------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY parity_gen IS
6     GENERIC (n: INTEGER := 7);   --default is 7
```

```
7     PORT (input: IN BIT_VECTOR(n DOWNTO 0);
8             output: OUT BIT_VECTOR(n+1 DOWNTO 0));
9  END parity_gen;
10 --------------------------------------------------
11 ARCHITECTURE parity OF parity_gen IS
12 BEGIN
13    PROCESS (input)
14       VARIABLE temp1: BIT;
15       VARIABLE temp2: BIT_VECTOR(output'RANGE);
16    BEGIN
17       temp1 := '0';
18       FOR i IN input'RANGE LOOP
19         temp1 := temp1 XOR input(i);
20         temp2(i) := input(i);
21       END LOOP;
22       temp2(output'HIGH) := temp1;
23       output <= temp2;
24    END PROCESS;
25 END parity;
26 --------------------------------------------------

1  ---------File my_code.vhd(actual project): -------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  --------------------------------------------------
5  ENTITY my_code IS
6     GENERIC(n: POSITIVE := 2);  -- 2 will overwrite 7
7     PORT (inp: IN BIT_VECTOR(n DOWNTO 0);
8             outp: OUT BIT_VECTOR(n+1 DOWNTO 0));
9  END my_code;
10 --------------------------------------------------
11 ARCHITECTURE my_arch OF my_code IS
12 --------------------------------------------------
13    COMPONENT parity_gen IS
14       GENERIC(n: POSITIVE);
15       PORT (input: IN BIT_VECTOR(n DOWNTO 0);
16             output: OUT BIT_VECTOR(n+1 DOWNTO 0));
17    END COMPONENT;
18 --------------------------------------------------
19 BEGIN
```

```
20    C1: parity_gen  GENERIC MAP(n) PORT MAP(inp, outp);
21 END my_arch;
22 ---------------------------------------------------------
```

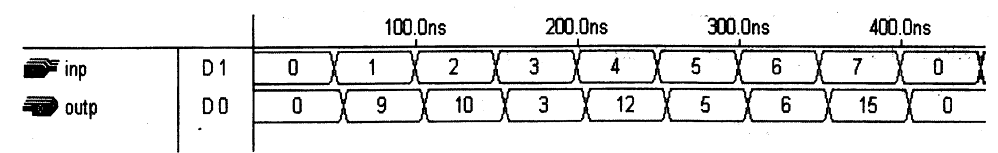
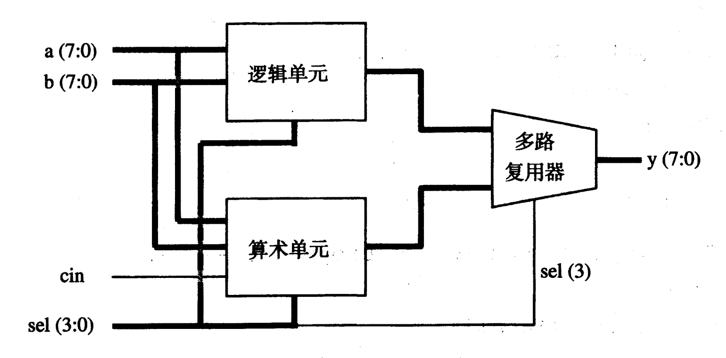


图 10.6　例 10.5 的仿真结果

例 10.6　通过元件实例化设计 ALU

例 5.5 中介绍了 ALU（算术逻辑单元）的设计方法（见图 10.7）。在当时的设计中，没有调用其他元件、函数或过程。在现在这个例子中，假设库文件中已经包含了构建 ALU 的 3 个元件：逻辑单元、算术单元和多路选择器。



sel	操作	功能	电路单元
0000	y<=a	选择 a	算术单元
0001	y<=a+1	将 a 加 1	
0010	y<=a–1	将 a 减 1	
0011	y<=b	选择 b	
0100	y<=b+1	将 b 加 1	
0101	y<=b–1	将 b 减 1	
0110	y<=a+b	将 a 与 b 相加	
0111	y<=a+b+cin	将 a，b 与进位相加	
1000	y<=NOTa	将 a 取反	逻辑单元
1001	y<=NOTb	将 b 取反	
1010	y<=a AND b	a 和 b 进行逻辑“与”运算	
1011	y<=a OR b	a 和 b 进行逻辑“或”运算	
1100	y<=a NAND b	a 和 b 进行逻辑“与非”运算	
1101	y<=a NOR b	a 和 b 进行逻辑“或非”运算	
1110	y<=a XOR b	a 和 b 进行逻辑“异或”运算	
1111	y<=a XNOR b	a 和 b 进行逻辑“同或”运算	

图 10.7　由 3 个元件构成的 ALU

在下面的代码中，除了代码的主体（alu.vhd）部分，还包括上面所述的3个元件。可以看到，元件在主代码中被声明。仿真结果见图10.8，与例5.5给出的仿真结果相似。

```
1  ----------COMPONENT arith_unit: -----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_unsigned.all;
5  -------------------------------------------------
6  ENTITY arith_unit IS
7       PORT (a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
8             sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
9             cin: IN STD_LOGIC;
10            x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
11 END arith_unit;
12 -------------------------------------------------
13 ARCHITECTURE arith_unit OF arith_unit IS
14    SIGNAL arith, logic: STD_LOGIC_VECTOR(7 DOWNTO 0);
15 BEGIN
16    WITH sel SELECT
17      x <=a WHEN "000",
18          a+1 WHEN "001",
19          a-1 WHEN "010",
20          b WHEN "011",
21          b+1 WHEN "100",
22          b-1 WHEN "101",
23          a+b WHEN "110",
24          a+b+cin WHEN OTHERS;
25 END arith_unit;
26 -------------------------------------------------
```

```
1  ----------COMPONENT logic_unit: -----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -------------------------------------------------
5  ENTITY logic_unit IS
6     PORT (a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7           sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
8           x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
9  END logic_unit;
```

```
10 ----------------------------------------------------
11 ARCHITECTURE logic_unit OF logic_unit IS
12 BEGIN
13 WITH sel SELECT
14     x <= NOT a WHEN "000",
15          NOT b WHEN "001",
16          a AND b WHEN "010",
17          a OR b WHEN "011",
18          a NAND b WHEN "100",
19          a NOR b WHEN "101",
20          a XOR b WHEN "110",
21          NOT(a XOR b) WHEN OTHERS;
22 END logic_unit;
23 ----------------------------------------------------
```

```
1  ------------COMPONENT mux: ---------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----------------------------------------------
5  ENTITY mux IS
6     PORT (a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7           sel: IN STD_LOGIC; -- (2 DOWNTO 0)
8           x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
9  END mux;
10 ----------------------------------------------------
11 ARCHITECTURE mux OF mux IS
12 BEGIN
13    WITH sel SELECT
14       x <= a WHEN '0',
15            b WHEN OTHERS;
16 END mux;
17 ----------------------------------------------------
```

```
1  ----------Project ALU (main code): --------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------------------
5  ENTITY alu IS
6     PORT (a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```
7           cin: IN STD_LOGIC;
8           sel: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9           y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
10 END alu;
11 --------------------------------------------------
12 ARCHITECTURE alu OF alu IS
13 --------------------------------------------------
14    COMPONENT arith_unit IS
15    PORT (a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
16          cin: IN STD_LOGIC;
17          sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
18          x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
19    END COMPONENT;
20    ----------------------------------------------
21    COMPONENT logic_unit IS
22    PORT (a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
23          sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
24          x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
25    END COMPONENT;
26    ----------------------------------------------
27    COMPONENT mux IS
28    PORT (a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
29          sel: IN STD_LOGIC;
30          x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
31    END COMPONENT;
32    ----------------------------------------------
33    SIGNAL x1, x2: STD_LOGIC_VECTOR (7 DOWNTO 0);
34 --------------------------------------------------
35 BEGIN
36    U1: arith_unit PORT MAP(a, b, cin, sel(2 DOWNTO 0), x1);
37    U2: logic_unit PORT MAP(a, b, sel(2 DOWNTO 0), x2);
38    U3: mux PORT MAP(x1, x2, sel(3), y);
39  END alu;
40  --------------------------------------------------
```

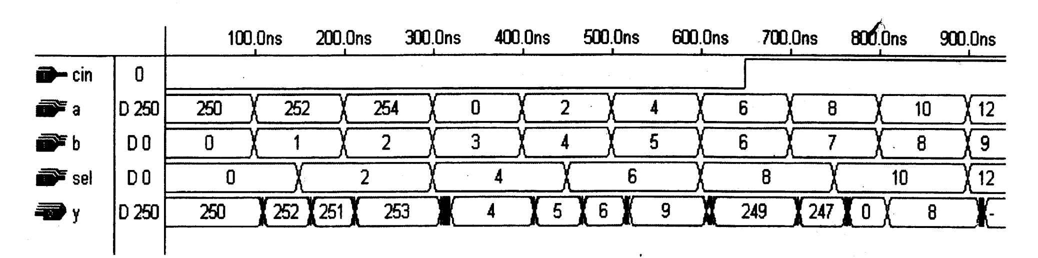


图 10.8 例 10.6 的仿真结果

10.6 习题

10.1 用包集中声明的元件构成 ALU。重新设计例 10.6 中的 ALU，这次需要建立一个包含所有元件的包集，然后对主代码进行必要的修改来完成设计。综合并仿真你的设计，验证其功能。

10.2 用元件构建逐级进位加法器。根据 9.3 节中讨论的逐级进位加法器（见图 9.6），设计一个全加器（FAU：Full Adder Unit），将它作为元件编译进 work 库，然后以实例化 FAU 的方式设计一个全加器。编译这些代码并进行综合与仿真，并将仿真结果与 9.3 节的结果进行比较。

10.3 用元件构建超前进位加法器。根据 9.3 节中的超前进位加法器（见图 9.8），设计一个 PGU 和 CLAU，并将它们作为元件编译进 work 库，然后通过实例化 PGU 和 CLAU 的方法设计超前进位加法器。可以选择在指定的包集中声明元件或在主代码中声明。编译代码并进行综合和仿真，将仿真结果与 9.3 节获得的结果进行比较。

10.4 寄存器输出的计数器。图 P10.4 给出了一个具有层次化结构的电路 STOP_WATCH，它由名为 COUNTER 和 REGISTER 的两个子电路（也就是元件）构成。只要 stop 有效，COUNTER 就复位一次，同时它的当前值被存储在 REGISTER 中，供用户观察数值。一旦 stop 返回'0'，COUNTER 就开始重新计数（从 0 开始），然而 REGISTER 中保持了先前存储的计数值。设计图 P10.4 所示的两个元件，然后在主代码中实例化它们，以完成 STOP_WATCH 电路。

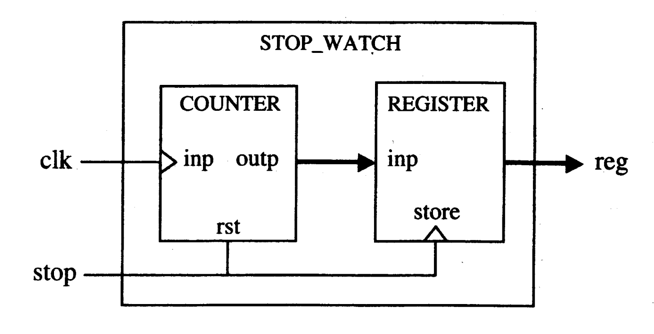


图 P10.4

第 11 章　函数和过程

函数（FUNCTION）和过程（PROCEDURE）统称为子程序。从结构特征上看，它们和第 6 章介绍过的进程（PROCESS）十分相似。它们内部包含的都是顺序描述的 VHDL 代码，通常使用相同的顺序描述语句，如 IF，CASE 和 LOOP（FUNCTION 和 PROCEDURE 中不允许使用 WAIT）。然而从应用的角度来看，PROCESS 与 FUNCTION 或 PROCEDURE 之间有着本质的区别。PROCESS 是直接在主代码段中使用的，而后者主要是为建库而使用的，它们的目的都是存储常用的 VHDL 代码，以达到代码重用和共享的目的。当然，如果需要，FUNCTION 和 PROCEDURE 也可以在主代码中直接建立并使用。

11.1　函数

一个函数（FUNCTION）就是一段顺序描述的代码。在代码编写过程中，有很多经常遇到的有共性的问题，如数据类型转换、逻辑运算操作、算术运算操作等。我们希望实现这些功能的代码可以被共享和重用，从而使主代码变得更简捷和易于理解，函数的建立和使用可以达到这个目的。

前面提到 FUNCTION 和 PROCESS（见 6.1 节）很相似，相同的语句（IF，WAIT，CASE 和 LOOP）既可用在进程中，也可用在函数中（WAIT 语句除外）。另外，在函数中禁止进行信号声明和元件实例化。

为了构建和使用函数，需要进行两个必要的步骤：函数体本身的创建和函数调用。

函数体

```
FUNCTION function_name [<parameter list>]RETURN data_type IS
    [声明]
BEGIN
    (顺序描述代码)
END function_name;
```

在上面的语法格式中，<parameter list>指出了函数的输入参数，输入参数可以是常量或信号，具体描述如下：

```
<parameter list >= [CONSTANT] 常量名：常量类型;
<parameter list >= SIGNAL 信号名：信号类型;
```

参数的个数可以是任意的，甚至不包含参数。变量不能作为参数。参数的类型可以是第 3 章

介绍的任意一种可综合的数据类型（布尔、标准逻辑、整数等），但不能指定它的取值范围。另一方面，函数只有一个返回值，这个返回值类型由 RETURN 后面的数据类型指定。

例 下面的名为 f1 的函数接收 3 个参数（a，b 和 c）。a 和 b 是常量（注意关键词 CONSTANT 可以省略），c 是信号。a 和 b 都是 INTEGER 类型的，而 c 是 STD_LOGIC_VECTOR 类型的，这里没有使用 RANGE 和 DOWNTO 等来约束输入参数的取值范围。输出参数（只能有一个）是 BOOLEAN 类型的。

```
FUNCTION f1(a, b: INTEGER; SIGNAL c: STD_LOGIC_VECTOR)
   RETURN BOOLEAN IS
BEGIN
   (顺序描述代码)
END f1;
```

函数调用

函数可以单独构成表达式，也可以作为表达式的一部分被调用。下面是函数调用的几个例子。

```
x <= conv_integer(a);           -- 将 a 转换为整型
                                -- 函数自身构成表达式
y <= maximum(a, b);             -- 返回 a 和 b 中较大的一个
                                -- 函数自身构成表达式
IF x>maximun(a, b)...           -- 将 x 与 a 和 b 中较大的一个进行比较
                                -- 函数作为表达式的一个组成部分
```

例 11.1 positive_edge( )函数

下面是一个时钟上升沿检测函数。它与 IF (clk'EVENT and clk = '1')语句实现类似的功能。在下面的一段代码中，该函数用于 D 触发器的设计。

```
---------------------函数体：------------------------
FUNCTION positive_edge(SIGNAL s: STD_LOGIC)RETURN BOOLEAN IS
BEGIN
   RETURN(s'EVENT AND s = '1');
END positive_edge;
---------------------函数调用：----------------------
...
IF positive_edge(clk) THEN...
...
---------------------------------------------------
```

例 11.2 conv_integer()函数

下面的函数将 STD_LOGIC_VECTOR 类型的数据转换为 INTEGER 类型。这段代码的通用性很强，可以处理任意宽度和方向（TO 或 DOWNTO）的输入矢量。该函数的典型调用方式如下所示：

```
------------------------函数-------------------------
FUNCTION conv_integer(SIGNAL vector: STD_LOGIC_VECTOR)
      RETUREN INTEGER IS
   VARIABLE result: INTEGER RANGE 0 TO 2 **vector'LENGTH-1;
BEGIN
    IF (vector(vector'HIGH) = '1') THEN result := 1;
    ELSE result := 0;
    END IF;
    FOR i IN (vector'HIGH-1) DOWNTO (vector'LOW)LOOP
        result := result*2;
        IF (vector(i) = '1') THEN result := result+1;
        END IF;
    END LOOP;
    RETURN result;
END conv_integer;
---------------------函数调用: ----------------------
...
y <= conv_integer(a);
...
----------------------------------------------------
```

11.2　函数的存放

函数和过程的典型存放位置如图 11.1 所示。虽然函数经常存放在包集中（目的是为了进行有效的代码分割、代码重用和代码共享），但也可以直接存放在主代码中（既可以存放在 ENTITY 中，也可以存放在 ARCHITECTURE 中）。

当函数被存放在 PACKAGE 中时，PACKAGE BODY 是必需的。PACKAGE BODY 中应存放在 PACKAGE 中所声明函数的函数体。下面对上述两种情况分别举例说明。

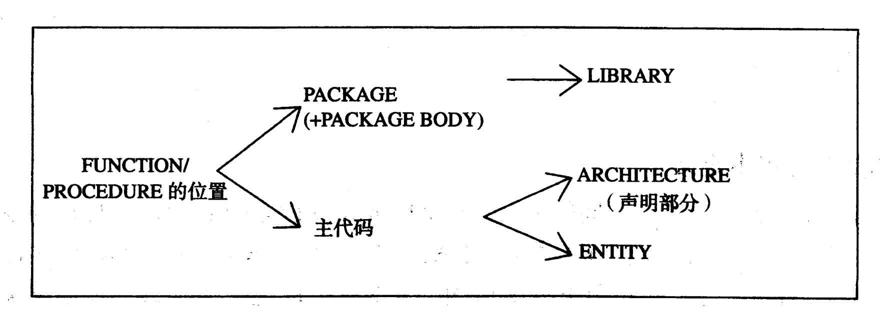


图 11.1　FUNCTION 和 PROCEDURE 的典型位置

例 11.3 函数在主代码中定义

以例 11.1 中的 positive_edge( )函数为例。正如上面提到的，当函数放在主代码中时，它既可以出现在 ENTITY 中，也可以出现在 ARCHITECTURE 中。在下面的例子中，函数放在 ARCHITECTURE 的声明部分，用来构建 D 触发器。

```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY dff IS
6     PORT (d, clk, rst: IN STD_LOGIC;
7           q: OUT STD_LOGIC);
8  END dff;
9  ------------------------------------------------
10 ARCHITECTURE my_arch OF dff IS
11 ------------------------------------------------
12 FUNCTION positive_edge (SIGNAL s: STD_LOGIC)
13    RETURN BOOLEAN IS
14 BEGIN
15   RETURN s'EVENT AND s = '1';
16 END positive_edge;
17 ------------------------------------------------
18 BEGIN
19   PROCESS (clk, rst)
20   BEGIN
21      IF (rst = '1') THEN q <= '0';
22      ELSIF positive_edge(clk) THEN q <= d;
23      END IF;
24   END PROCESS;
25 END my_arch;
26 ------------------------------------------------
```

例 11.4 函数在包集中定义

本例题与例 11.3 相似，惟一不同的是函数在这里被放在 PACKAGE 中，可以方便地被其他设计所重用和共享。当放在包集中时，函数在 PACKAGE 中声明，函数体出现在 PACKAGE BODY 中。

下面有两段 VHDL 代码，一段用于构建 FUNCTION/PACKAGE，另一段是调用该函数的例子。两段代码可以被分别编译成两个独立的文件，也可以被编译成一个单独的文件（保存为 dff.vhd，它是 ENTITY 的名字）。注意，在主代码中包含了 USE work.my_package.all; 语句。

```
1   ---------------Package: ------------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   -------------------------------------------------
5   PACKAGE my_package IS
6      FUNCTION positive_edge(SIGNAL s: STD_LOGIC)RETURN BOOLEAN;
7   END my_package;
8   -------------------------------------------------
9   PACKAGE BODY my_package IS
10     FUNCTION positive_edge(SIGNAL s: STD_LOGIC)
11        RETURN BOOLEAN IS
12     BEGIN
13        RETURN s'EVENT AND s = '1';
14     END positive_edge;
15  END my_package;
16  -------------------------------------------------
1   --------------------Main code: ------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   USE work.my_package.all;
5   -------------------------------------------------
6   ENTITY dff IS
7      PORT (d, clk, rst: IN STD_LOGIC;
8            q: OUT STD_LOGIC);
9   END dff;
10  -------------------------------------------------
11  ARCHITECTURE my_arch OF dff IS
12  BEGIN
13     PROCESS (clk, rst)
14     BEGIN
15        IF (rst = '1') THEN q <= '0';
16        ELSIF positive_edge(clk) THEN q <= d;
17        END IF;
18     END PROCESS;
19  END my_arch;
20 -------------------------------------------------
```

例 11.5　conv_integer( )函数

函数 conv_integer( )在例 11.2 中已经出现过，它可以将 STD_LOGIC_VECTOR 类型的数值转换为整数类型的数值。这里将函数放进 PACKAGE 中，在主代码中将调用这个函数。

```
1   ------------Package: ----------------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   --------------------------------------------------
5   PACKAGE my_package IS
6   FUNCTION conv_integer(SIGNAL vector: STD_LOGIC_VECTOR)
7      RETURN INTEGER ;
8   END my_package;
9   --------------------------------------------------
10  PACKAGE BODY my_package IS
11     FUNCTION conv_integer (SIGNAL vector: STD_LOGIC_VECTOR)
12           RETURN INTEGER IS
13        VARIABLE result: INTEGER RANGE 0 TO 2**vector'LENGTH-1;
14     BEGIN
15        IF (vector(vector'HIGH) = '1') THEN result := 1;
16        ELSE result := 0;
17        END IF;
18        FOR i IN (vector'HIGH-1)DOWNTO (vector'LOW) LOOP
19           result := result*2;
20           IF (vector(i) = '1') THEN result := result+1;
21           END IF;
22        END LOOP;
23        RETURN result;
24     END conv_integer;
25  END my_package;
26  --------------------------------------------------
```

```
1   ---------------Main code: -----------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   USE work.my_package.all;
5   --------------------------------------------------
6   ENTITY conv_int2 IS
7      PORT (a: IN STD_LOGIC_VECTOR(0 TO 3);
8            y: OUT INTEGER RANGE 0 TO 15);
9   END conv_int2;
10  --------------------------------------------------
11  ARCHITECTURE my_arch OF conv_int2 IS
12  BEGIN
```

```
13      y <= conv_integer(a);
14   END my_arch;
15   --------------------------------------------------
```

例 11.6　“+”函数

下面给出的函数称为“+”函数，它扩展了先前定义的“+”操作符。预定义的“+”操作符只能允许 INTEGER，SIGNED 和 UNSIGNED 类型的数据之间的运算。然而，我们希望编写一个能够进行 STD_LOGIC_VECTOR 类型数据加法运算的函数。

下面给出的函数被存放在 PACKAGE 和相应的 PACKAGE BODY 中。后面紧接着给出了调用该函数的代码。需要注意的是，传递给函数的两个参数与函数的返回值都是 STD_LOGIC_VECTOR 类型的矢量，并且它们都具有相同的位宽。

```
1    ----------------Package: ------------------------
2    LIBRARY ieee;
3    USE ieee.std_logic_1164.all;
4    --------------------------------------------------
5    PACKAGE my_package IS
6       FUNCTION "+" (a, b: STD_LOGIC_VECTOR)
7             RETURN STD_LOGIC_VECTOR;
8    END my_package;
9    --------------------------------------------------
10   PACKAGE BODY my_package IS
11      FUNCTION "+" (a, b: STD_LOGIC_VECTOR)
12            RETURN STD_LOGIC_VECTOR IS
13        Variable result: STD_LOGIC_VECTOR(3 DOWNTO 0);
14        Variable carry: STD_LOGIC;
15      BEGIN
16        carry := '0';
17        FOR i IN a'REVERSE_RANGE LOOP
18          result(i) := a(i) XOR b(i) XOR carry;
19          carry := (a(i) AND b(i))OR(a(i)AND carry)OR
20                   (b(i)AND carry);
21        END LOOP;
22        RETURN result;
23      END "+";
24   END my_package;
25   --------------------------------------------------

1    -------------------Main code: -----------------
```

```
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   USE work.my_package.all;
5   ---------------------------------------------
6   ENTITY add_bit IS
7      PORT (a: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
8            y: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
9   END add_bit;
10  ---------------------------------------------
11  ARCHITECTURE my_arch OF add_bit IS
12     CONSTANT b: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0011";
13     CONSTANT C: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0110";
14  BEGIN
15     y <= a+b+c;     ---overloaded"+"operator
16  END my_arch;
17  ---------------------------------------------
```

图 11.2 中给出的是当位宽为 4 时的仿真结果。我们将两个常量 b = 3 和 c = 6 与输入信号 a 进行 “+” 运算，期望的结果是 y = a + 9。

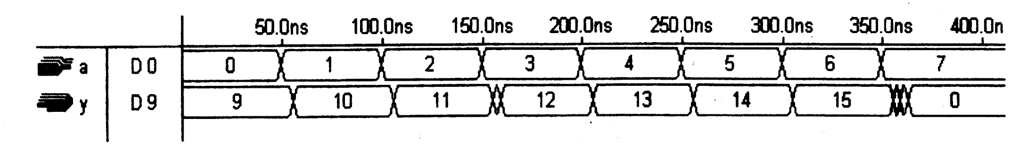


图 11.2　例 11.6 的仿真结果

例 11.7　算术移位函数

下面给出的函数能够将 STD_LOGIC_VECTOR 类型的矢量进行算术左移运算。传递给函数的两个参数为 arg1 和 arg2，是需要进行移位操作的矢量和移动的位数。由于该函数具有 GENERIC 参数（见第 13 行~第 26 行），对于任意位宽和方向的输入矢量都可以处理。在这个例子中，函数出现在主代码中，而不是出现在 PACKAGE 中。

```
1   ----------------Package: ------------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   ---------------------------------------------
5   ENTITY shift_left IS
6      GENERIC (size: INTEGER := 4);
7      PORT (a: IN STD_LOGIC_VECTOR(size-1 DOWNTO 0);
```

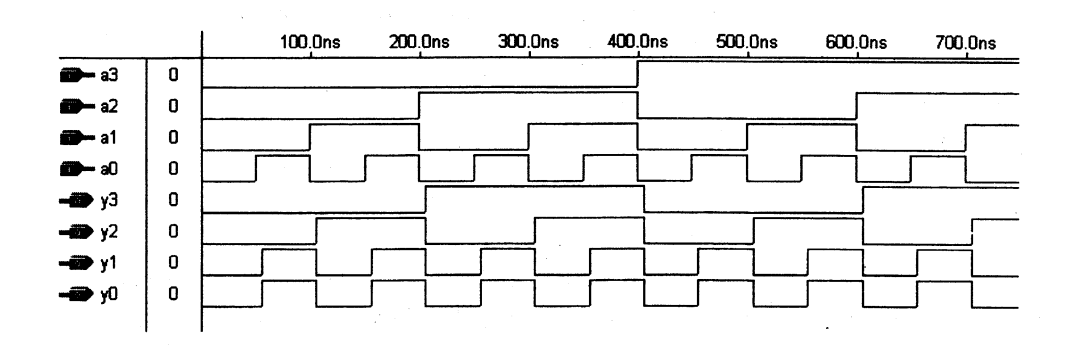
```
8           x, y, z: OUT STD_LOGIC_VECTOR(size-1 DOWNTO 0));
9    END shift_left;
10   -------------------------------------------------
11   ARCHITECTURE behavior OF shift_left IS
12   -------------------------------------------------
13     FUNCTION slar (arg1: STD_LOGIC_VECTOR; arg2: NATURAL)
14           RETURN STD_LOGIC_VECTOR IS
15        VARIABLE input: STD_LOGIC_VECTOR(size-1 DOWNTO 0) := arg1;
16        CONSTANT size: INTEGER := arg1'LENGTH;
17        VARIABLE copy: STD_LOGIC_VECTOR(size-1 DOWNTO 0)
18           := (OTHERS => arg1(arg1'RIGHT));
19        VARIABLE result: STD_LOGIC_VECTOR(size-1 DOWNTO 0);
20     BEGIN
21        IF (arg2 >= size-1) THEN result  := copy;
22        ELSE result  :=  input(size-1-arg2 DOWNTO 1)&
23           copy(arg2 DOWNTO 0);
24        END IF;
25        RETURN result;
26     END slar;
27   -------------------------------------------------
28   BEGIN
29     x <= slar(a, 0);
30     y <= slar(a, 1);
31     z <= slar(a, 2);
32   END behavior;
33   -------------------------------------------------
```

仿真结果见图 11.3。在图中上半部的仿真波形中，输入矢量是 a(3 DOWNTO 0)，a(3)是最高位（见第 7 行）；在下面的仿真波形中，输入矢量是 a(0 TO 3)，a(0)是最高位。

例 11.8　乘法器

在这个例子中，函数的名称为 mult( )。它可以对两个 UNSIGNED 类型的数据进行乘法运算，并返回 UNSIGNED 类型的结果。传递给函数的参数可以具有不同的位宽，它们的方向（TO/DOWNTO）可以是任意的。函数存放在名为 pack 的 PACKAGE 中。后面给出了调用这个函数的例子。仿真结果见图 11.4。

```
1    ---------------Package: ------------------------
2    LIBRARY ieee;
```



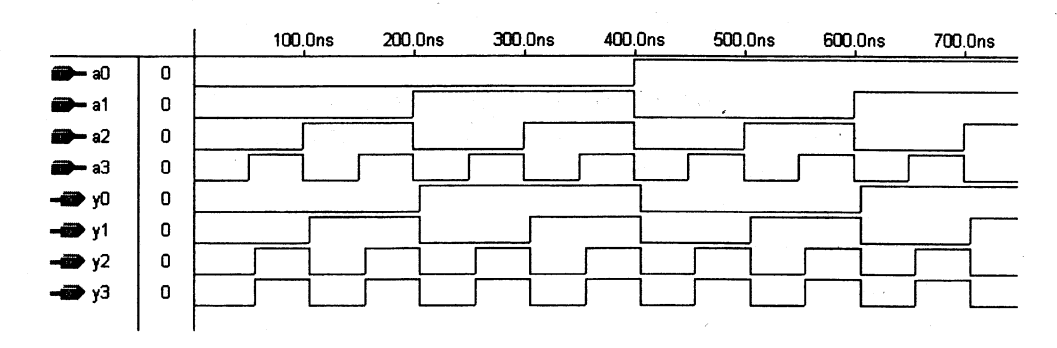


图 11.3 例 11.7 的仿真结果

```
3   USE ieee.std_logic_1164.all;
4   USE ieee.std_logic_arith.all;
5   --------------------------------------------------
6   PACKAGE pack IS
7      FUNCTION mult (a, b: UNSIGNED)RETURN UNSIGNED;
8   END pack;
9   --------------------------------------------------
10  PACKAGE BODY pack IS
11     FUNCTION mult(a, b: UNSIGNED)RETURN UNSIGNED IS
12        CONSTANT max: INTEGER  :=  a'LENGTH+b'LENGTH-1;
13        VARIABLE aa: UNSIGNED(max DOWNTO 0)  :=
14           (max DOWNTO a'LENGTH => '0')
15           &a(a'LENGTH-1 DOWNTO 0);
16        VARIABLE prod: UNSIGNED(max DOWNTO 0)  :=  (OTHERS => '0');
17     BEGIN
18        FOR i IN 0 TO a'LENGTH-1 LOOP
```

```
19            IF (b(i) = '1') THEN prod  :=  prod+aa;
20            END IF;
21            aa  :=  aa(max-1 DOWNTO 0)&'0';
22         END LOOP;
23         RETURN prod;
24      END mult;
25   END pack;
26   --------------------------------------------------
```

```
1    -----------------Main code: ---------------------
2    LIBRARY ieee;
3    USE ieee.std_logic_1164.all;
4    USE ieee.std_logic_arith.all;
5    USE work.my_package.all;
6    --------------------------------------------------
7    ENTITY multiplier IS
8       GENERIC(size: INTEGER  :=  4);
9       PORT (a, b: IN UNSIGNED(size-1 DOWNTO 0);
10              y: OUT UNSIGNED(2*size-1 DOWNTO 0));
11   END multiplier;
12   --------------------------------------------------
13   ARCHITECTURE behavior OF multiplier IS
14   BEGIN
15      y <= mult(a, b);
16   END behavior;
17   --------------------------------------------------
```

		500.0ns	1.0us	1.5us	2.0us	2.5us	3.0us	3.5us	4.0us	4.5us
a	D0	0, 2	4	6	8	10	12	14	0	2
b	D0	0	5		10		15		4	
y	D0	0	20	30	80	100	180	210	0	8

图 11.4　例 11.8 的仿真结果

11.3　过程

过程与函数相似，其目的也相同，它们的主要差别在于过程可以具有多个返回值。与函数类似，过程的定义和使用包括两个部分：过程本身的定义和过程调用。

```
PROCEDURE 过程名 [<参数列表>] IS
    [声明]
BEGIN
    (顺序描述语句)
END 过程名;
```

在上面的语法格式中，参数列表指出了过程的输入和输出参数，具有如下的格式：

<parameter list >= [CONSTANT] 常量名：模式类型；

<parameter list >= SIGNAL 信号名：模式类型；

<parameter list >= VARIABLE 变量名：模式类型；

过程的参数可以有任意多个，参数可以是 IN，OUT 或 INOUT 模式的信号、变量或常量。对于输入模式（IN）的参数，默认情况下为常量，而对于输出模式（OUT 或 INOUT）的参数，默认情况下为变量。

通过前面章节的学习，可以知道函数内部的 WAIT 语句、信号声明和元件调用是不可综合的。对于过程来说也是如此，惟一不同的是，如果一个过程在一个 PROCESS 中进行了声明，那么其内部可以声明信号。此外，除了 WAIT 语句以外的任何边沿检测在过程中都是不可综合的（也就是说，与函数相比较，一个可综合的过程内部不能包含或隐含寄存器）。

在 11.5 节中将对函数和过程进行对比和总结。

例 下面的过程有 3 个输入信号：a，b 和 c。a 是 BIT 类型的常量，而 b 和 c 是信号，同样是 BIT 类型的。注意，对于输入参数来说，CONSTANT 这个词是可以省略的。这里同样有两个返回信号 x（OUT，BIT_VECTOR）和 y（INOUT，INTEGER）。

```
PROCEDURE my_procedure( a: IN BIT; SIGNAL b, c: IN BIT;
                        SIGNAL x: OUT BIT_VECTOR(7 DOWNTO 0);
                        SIGNAL y: INOUT INTEGER RANGE 0 TO 99) IS
BEGIN
    ...
END my_procedure;
```

过程调用

与作为表达式一部分的函数调用相比，过程调用的形式就显得更简单一些。

下面是几个过程调用的例子：

```
compute_min_max(in1, in2, in3, out1, out2);
            -- 直接进行过程调用
divide(dividend, divisor, quotient, remainder);
            -- 直接进行过程调用
```

```
IF (a>b) THEN compute_min_max(in1, in2, in3, out1, out2);
                 -- 在其他语句中进行过程调用
```

11.4　过程的存放

过程代码的存放位置与函数类似（见图 11.1）。为了有利于代码的分割、重用与共享，过程经常存放在 PACKAGE 中，当然过程也可以存放在主代码中（在 ENTITY 或 ARCHITECTURE 的声明部分）。如果放置在 PACKAGE 中，那么相应的 PACKAGE BODY 一定是需要的，因为在 PACKAGE 中已声明了过程的主体代码需要存放在 PACKAGE BODY 中。后面对这两种方式分别进行了举例说明。

例 11.9　过程存放在主代码中

下面的 min_mix 代码使用了名为 sort 的过程。sort 的输入参数是两个 8 位的无符号整数（inp1 和 inp2）。过程对它们进行了比较，数值小的从 min_out 输出，数值大的从 max_out 输出（见图 11.5）。过程位于 ARCHITECTURE 的声明部分。这里应该注意过程的调用方式。电路的仿真结果见图 11.6。

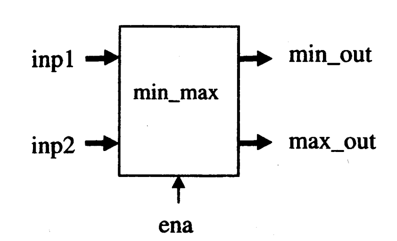


图 11.5　例 11.9 中的 min_max 电路

图 11.6　例 11.9 中的仿真结果

```
1   ---------------------------------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   ---------------------------------------------
5   ENTITY min_max IS
6      GENERIC(limit: INTEGER  :=  255);
7      PORT (ena: IN BIT;
```

```
8            inp1, inp2: IN INTEGER RANGE 0 TO limit;
9            min_out, max_out: OUT INTEGER RANGE 0 TO limit);
10   END min_max;
11   ------------------------------------------------
12   ARCHITECTURE my_architecture OF min_max IS
13      ---------------------------------------------
14      PROCEDURE sort (SIGNAL in1, in2: IN INTEGER RANGE 0 TO limit;
15         SIGNAL min, max: OUT INTEGER RANGE 0 TO limit) IS
16      BEGIN
17         IF (in1>in2) THEN
18            max <= in1;
19            min <= in2;
20            ELSE
21            max <= in2;
22            min <= in1;
23            END IF;
24      END sort;
25     ----------------------------------------------
26   BEGIN
27      PROCESS (ena)
28      BEGIN
29            IF (ena = '1') THEN sort (inp1, inp2, min_out, max_out);
30            END IF;
31      END PROCESS;
32   END my_architecture;
33   ------------------------------------------------
```

例 11.10 PACKAGE 中的过程

本例题与上一例题相似，不同之处在于过程 sort 被存放在名为 my_package 的包集中，以便于被其他设计重用和共享。下面的代码可以编译成两个分开的文件或者一个单一的文件（称为 min_max.vhd）。

```
1    ---------------Package:-------------------------
2    LIBRARY ieee;
3    USE ieee.std_logic_1164.all;
4    ------------------------------------------------
5    PACKAGE my_package IS
6       CONSTANT limit: INTEGER := 255;
7       PROCEDURE sort (SIGNAL in1, in2: IN INTEGER RANGE 0 TO limit;
```

```
8         SIGNAL min, max: OUT INTEGER RANGE 0 TO limit);
9     END my_package;
10    -------------------------------------------------
11    PACKAGE BODY my_package IS
12       PROCEDURE sort (SIGNAL in1, in2: IN INTEGER RANGE 0 TO limit;
13          SIGNAL min, max: OUT INTEGER RANGE 0 TO limit) IS
14       BEGIN
15          IF (in1>in2) THEN
16             max <= in1;
17             min <= in2;
18          ELSE
19             max <= in2;
20             min <= in1;
21          END IF;
22        END sort;
23    END my_package;
24    -------------------------------------------------
```

```
1     ----------------Main code:-----------------------
2     LIBRARY ieee;
3     USE ieee.std_logic_1164.all;
4     USE work.my_package.all;
5     -------------------------------------------------
6     ENTITY min_max1 IS
7        GENERIC(limit: INTEGER := 255);
8          PORT (ena: IN BIT;
9                inp1, inp2: INTEGER RANGE 0 TO limit;
10               min_out, max_out: OUT INTEGER RANGE 0 TO limit);
11    END min_max1;
12    -------------------------------------------------
13    ARCHITECTURE my_architecture OF min_max1 IS
14    BEGIN
15       PROCESS (ena)
16       BEGIN
17          IF (ena = '1') THEN sort (inp1, inp2, min_out, max_out);
18          END IF;
19       END PROCESS;
20    END my_architecture;
21    -------------------------------------------------
```

11.5 函数与过程小结

- 函数有零个或多个输入参数和一个返回值。输入参数只能是常量（默认）或信号（不允许是变量）。
- 过程可以带有多个输入、输出或双向参数。这些参数可以是信号、变量或常量。对于输入模式（IN）的参数，默认情况下为常量，而对于输出模式（OUT 或 INOUT）的参数，默认情况下为变量。
- 函数调用是作为表达式的一部分出现的，过程的调用相对而言更简单，可以直接进行调用。
- 在函数和过程的内部，WAIT 和 COMPONENTS 都是不可综合的。
- 函数和过程的存放位置是相同的（见图 11.1）。它们经常位于 PACKAGE 中或主代码中（在 ENTITY 或 ARCHITECTURE 中）。当位于 PACKAGE 中时，对应的 PACKAGE BODY 必须存在，其中存放着函数或过程的功能描述代码。

11.6 断言语句

ASSERT 语句是不可综合的，它的作用是将仿真过程中发现的问题通过屏幕显示等方法指出来。根据问题的严重程度，仿真过程可以被命令终止。其语法格式如下：

```
ASSERT condition
[REPORT "message"]
[SEVERITY severity_level];
```

严重程度的等级可以划分为：注意、警告、错误和失败，其中“错误”是默认的。当判断条件（condition）值为假（FALSE）时，就会显示 message。

例 我们要写一个函数来进行两个二进制数相加的运算（如例 11.6 所示），这里要求两个输入参数必须具有相同的位宽。为了检测这个要求是否得到满足，可以在函数体内加入下面的 ASSERT 语句。

```
ASSERT a'LENGTH = b'LENGTH
REPORT "Error: vectors do not have same length!"
SEVERITY failure;
```

此外，ASSERT 语句不会消耗硬件资源。综合工具会将其忽略或者给出警告，指出其不可综合。

11.7 习题

11.1 **STD_LOGIC_VECTOR 类型转换**。编写类型转换函数 conv_std_logic( )，它能够将 INTEGER 类型的数据转换为 STD_LOGIC_VECTOR 类型的矢量。编写一个调用该函数的实例，验证

设计的正确性。用两种方法实现整个设计：一种方法是将函数存放在主代码中，另一种方法是将其存放在 PACKAGE 中。

11.2　NOT 运算符扩展。NOT 运算符可以对二进制数进行取反运算。例如，x = "1000"是 STD_LOGIC_VECTOR 类型的矢量，那么 NOT x 的结果是"0111"。如果 x 是 INTEGER 类型的，就不允许进行取反运算。编写一个名为 not 的函数来实现对 INTEGER 类型数据的取反运算（见 4.4 节和例 11.6）。

11.3　STD_LOGIC_VECTOR 类型矢量的逻辑移位操作。预定义的移位操作符（见 4.1 节）的操作对象是 BIT_VECTOR 类型的矢量。编写一个函数，它可以将 STD_LOGIC_VECTOR 类型的矢量根据指定的数值向左进行逻辑移位。该函数需要传递两个参数：需要进行移位操作的 STD_LOGIC_VECTOR 类型的矢量本身和一个指定左移位数的 NATURAL 类型的数值。将函数放入 PACKAGE 中，然后编写一个调用该函数的例子，并对设计进行验证（建议：回顾例 11.7）。

11.4　整数的逻辑移位。本习题是习题 11.3 的推广。编写一个函数，可以根据给定的数值将一个整数进行左移操作。将该函数放在一个包集中，然后编写一个调用该函数的例子，以进行设计验证。

11.5　有符号乘法器。编写一个和例 11.8 相似的函数，这个函数的输入和输出都是有符号数。

11.6　带有 SSD 显示功能的两位十进制数计数器。例 6.7 是一个两位的十进制计数器，它的计数范围是 0→99→0。该计数器具有异步复位端和将二进制编码的十进制数（BCD）转换为 7 段显示器（SSD）驱动信号的转换电路，其中将 BCD 转换为 SSD 驱动信号的代码使用了两次。如果使用函数，那么代码的结构会更清晰。编写实现该功能的函数 bcd_to_ssd，并将其存放到包集中。以函数调用的方式重新设计例 6.7，然后对代码进行综合和仿真。

11.7　编写用于数值统计的 PROCEDURE。编写一个 PROCEDURE，它接收 8 个有符号的值，返回其平均值（ave）、最大值（max）和最小值（min）。PROCEDURE 被存放在包集中。编写一个调用该 PROCEDURE 的例子对其进行测试。

11.8　"+"操作符扩展。在例 11.6 中介绍了一个对"+"操作符进行扩展的函数。它的目的是允许直接对 STD_LOGIC_VECTOR 类型的数进行加法运算。在例 11.6 中，返回的参数与输入参数具有相同的位宽。编写一个类似的函数，在返回的矢量中增加一位，它与进位位功能类似，通过查看该位可以很容易地检测到运算是否溢出。

第 12 章　系统设计实例分析

本章将分析多个有代表性的电路，目的是应用前面章节学习的与系统级电路设计相关的方法和技巧。在这些例子中普遍使用了包集、元件、函数和过程。

12.1　串–并型乘法器

图 12.1 给出了串–并型乘法器的详细电路结构图。输入矢量 a 以串行的方式，从低位开始，逐位输送到运算电路中。另一个输入矢量 b 的所有位以并行的方式同时输入。假设 a 有 M 位，b 有 N 位，那么当 a 的 M 位都进入系统之后，其后要填充 N 个'0'，使输入达到 M+N 位，这也是最终乘法运算输出结果的位宽。

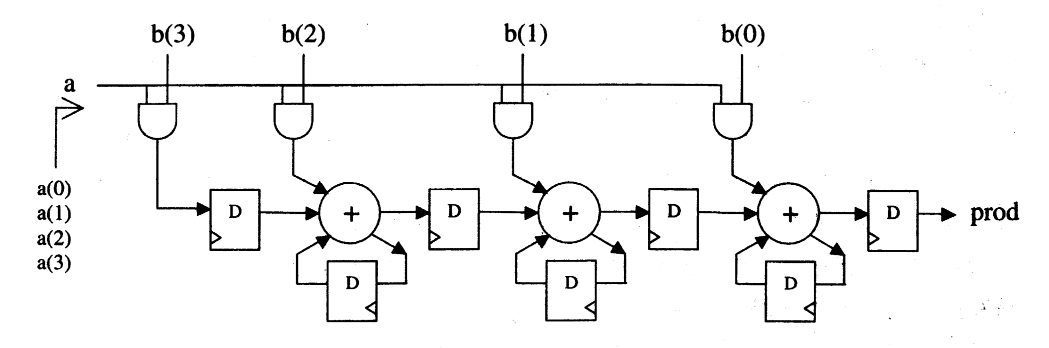


图 12.1　串–并型乘法器

由图 12.1 可知，系统采用的是流水线结构，整个电路由与门、全加器单元和触发器构成。除了最左边的一个，每个流水线单元都需要一个加法器、两个寄存器和一个与门来计算其中的一位输入。

下面给出的代码采用的是结构化的描述方法，也就是说，整个设计由多个元件组合而成。这里的元件实例化不止一级，pipe 是被顶层电路实例化的元件，其本身也是通过实例化其他元件而得到的。

下面给出了每个元件的设计代码以及包括所有元件声明的包集。可以看出主代码的结构非常清晰和简捷。整个电路的仿真结果在图 12.2 中给出。

```
1 ---------------and_2.vhd(component): -------------
2 LIBRARY ieee;
```

```
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  ENTITY and_2 IS
6     PORT (a, b: IN STD_LOGIC;
7            y: OUT STD_LOGIC);
8  END and_2;
9  ---------------------------------------------------
10 ARCHITECTURE and_2 OF and_2 IS
11 BEGIN
12     y <= a AND b;
13 END and_2;
14 ---------------------------------------------------
```

```
1  ----------------reg.vhd(component): - -----------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4     -----------------------------------------------
5  ENTITY reg IS
6     PORT (d, clk, rst: IN STD_LOGIC;
7            q: OUT STD_LOGIC);
8  END reg;
9  ---------------------------------------------------
10 ARCHITECTURE reg OF reg IS
11 BEGIN
12     PROCESS (clk, rst)
13     BEGIN
14        IF (rst = '1') THEN q <= '0';
15        ELSIF (clk'EVENT AND clk = '1') THEN q <= d;
16        END IF;
17    END PROCESS;
18 END reg;
19 ---------------------------------------------------
```

```
1  -----------------fau.vhd(component): ------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  ENTITY fau IS
6     PORT (a, b, cin: IN STD_LOGIC;
7            s, cout: OUT STD_LOGIC);
```

```
8  END fau;
9  ---------------------------------------------------
10 ARCHITECTURE fau OF fau IS
11 BEGIN
12    s <= a XOR b XOR cin;
13    cout <= (a AND b)OR(a AND cin)OR (b AND cin);
14 END fau;
15 ---------------------------------------------------
```

```
1  ----pipe.vhd (component): -----------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_components.all;
5  ---------------------------------------------------
6  ENTITY pipe IS
7     PORT (a, b, clk, rst: IN STD_LOGIC;
8           q: OUT STD_LOGIC);
9  END pipe;
10 ---------------------------------------------------
11 ARCHITECTURE structural OF pipe IS
12    SIGNAL s, cin, cout: STD_LOGIC;
13 BEGIN
14    U1: COMPONENT fau PORT MAP(a, b, cin, s, cout);
15    U2: COMPONENT reg PORT MAP(cout, clk, rst, cin);
16    U3: COMPONENT reg PORT MAP(s, clk, rst, q);
17 END structural;
18 ---------------------------------------------------
```

```
1  -----------my_components.vhd (package): ---------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  PACKAGE my_components IS
6  ---------------------------------------------------
7  COMPONENT and_2 IS
8     PORT (a, b: IN STD_LOGIC; y: OUT STD_LOGIC);
9  END COMPONENT;
10 ---------------------------------------------------
11  COMPONENT fau IS
12     PORT (a, b, cin: IN STD_LOGIC; s, cout: OUT STD_LOGIC);
```

```
13  END  COMPONENT;
14  ---------------------------------------------------
15  COMPONENT reg IS
16     PORT (d, clk, rst: IN STD_LOGIC; q: OUT STD_LOGIC);
17  END COMPONENT;
18  ---------------------------------------------------
19  COMPONENT Pipe IS
20     PORT (a, b, clk, rst: IN STD_LOGIC; q: OUT STD_LOGIC);
21  END  COMPONENT;
22  ---------------------------------------------------
23  END my_components;
24  ---------------------------------------------------
```

```
1  ----multiplier.vhd(project): --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_components.all;
5  ---------------------------------------------------
6  ENTITY multiplier IS
7     PORT (a, clk, rst: IN STD_LOGIC;
8           b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9           prod: OUT STD_LOGIC);
10 END multiplier;
11 ---------------------------------------------------
12 ARCHITECTURE structural OF multiplier IS
13    SIGNAL and_out, reg_out: STD_LOGIC_VECTOR(3 DOWNTO 0);
14 BEGIN
15    U1: COMPONENT and_2 PORT MAP (a, b(3), and_out(3));
16    U2: COMPONENT and_2 PORT MAP (a, b(2), and_out(2));
17    U3: COMPONENT and_2 PORT MAP (a, b(1), and_out(1));
18    U4: COMPONENT and_2 PORT MAP (a, b(0), and_out(0));
19    U5: COMPONENT reg PORT MAP (and_out(3), clk, rst,
20       reg_out(3));
21    U6: COMPONENT pipe PORT MAP (and_out(2), reg_out(3),
22       clk, rst, reg_out(2));
23    U7: COMPONENT pipe PORT MAP (and_out(1), reg_out(2),
24       clk, rst, reg_out(1));
25    U8: COMPONENT pipe PORT MAP (and_out(0), reg_out(1),
26       clk, rst, reg_out(0));
```

```
27    prod <= reg_out(0);
28 END structural;
29 -------------------------------------------------
```

图 12.2　串–并型乘法器的仿真结果

根据图 12.2 的仿真结果可以看到，a = "1100"（十进制 12）从低位开始串行输入。最低位 a(0) 出现在 100~200 ns 之间，而最高位 a(3)出现在 400~500 ns 之间，此后跟随的是 4 个连续的'0'。另一个信号 b = "1101"（十进制 13）以并行的方式输入。乘法运算的结果 prod = "10011100"（十进制的 156）在图的最下面给出，首先计算得到的第一位是最低位 prod(0) = '0'，它出现在第一个时钟上升沿之后，在 150~250 ns 之间，而 prod 的最后一位（最高位）出现在 850~950 ns 之间。

12.2　并行乘法器

图 12.3 给出的是 4 位并行乘法器的电路结构图。与图 12.1 比较，并行乘法器的两个操作数都是并行输入的，因此不需要使用寄存器进行存储。从图 12.3 中可以看出，它由与门和全加器单元（FAU）构成。它的操作数是 a 和 b（每个都是 4 位宽度），结果是 prod（宽度为 8 位）。

下面给出的代码是基于元件实例化的。这里首先给出了两个基本元件：AND_2 和 FAU（见 12.1 节）。top_row，mid_row 和 lower_row 是通过实例化基本元件得到的。这些元件都在包集 my_component 中声明，主代码 multiplier 调用这些元件，最终完成了整个设计（见图 12.3）。图 12.4 给出了电路的仿真波形。

```
1  --------top_row.vhd(component): ----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_components.all;
5  -------------------------------------------------
6  ENTITY top_row IS
7     PORT (a: IN STD_LOGIC;
8           b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9           sout, cout: OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
10          p: OUT STD_LOGIC);
11 END top_row;
```

```
12 -------------------------------------------------
13 ARCHITECTURE structural OF top_row IS
14 BEGIN
15    U1: COMPONENT and_2 PORT MAP(a, b(3), sout(2));
16    U2: COMPONENT and_2 PORT MAP(a, b(2), sout(1));
17    U3: COMPONENT and_2 PORT MAP(a, b(1), sout(0));
18    U4: COMPONENT and_2 PORT MAP(a, b(0), p);
19    cout(2) <= '0'; cout(1) <= '0'; cout(0) <= '0';
20 END structural;
21 -------------------------------------------------
```

```
1  ---------mid_row.vhd(component): ----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_components.all;
5  -------------------------------------------------
6  ENTITY mid_row IS
7     PORT (a: IN STD_LOGIC;
8           b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9           sin, cin: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
10          sout, cout: OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
11          p: OUT STD_LOGIC);
12 END top_row;
13 -------------------------------------------------
14 ARCHITECTURE structural OF mid_row IS
15    SIGNAL and_out: STD_LOGIC_VECTOR(2 DOWNTO 0);
16 BEGIN
17    U1: COMPONENT and_2 PORT MAP(a, b(3), sout(2));
18    U2: COMPONENT and_2 PORT MAP(a, b(2), and_out(2));
19    U3: COMPONENT and_2 PORT MAP(a, b(1), and_out(1));
20    U4: COMPONENT and_2 PORT MAP(a, b(0), and_out(0));
21    U5: COMPONENT fau PORT MAP(sin(2), cin(2), and_out(2)
22       sout(1), cout(2));
23    U6: COMPONENT fau PORT MAP(sin(1), cin(1), and_out(1)
24       sout(0), cout(1));
25    U7: COMPONENT fau PORT MAP(sin(0), cin(0), and_out(0)
26       p, cout(0));
27  END structural;
28 -------------------------------------------------
```

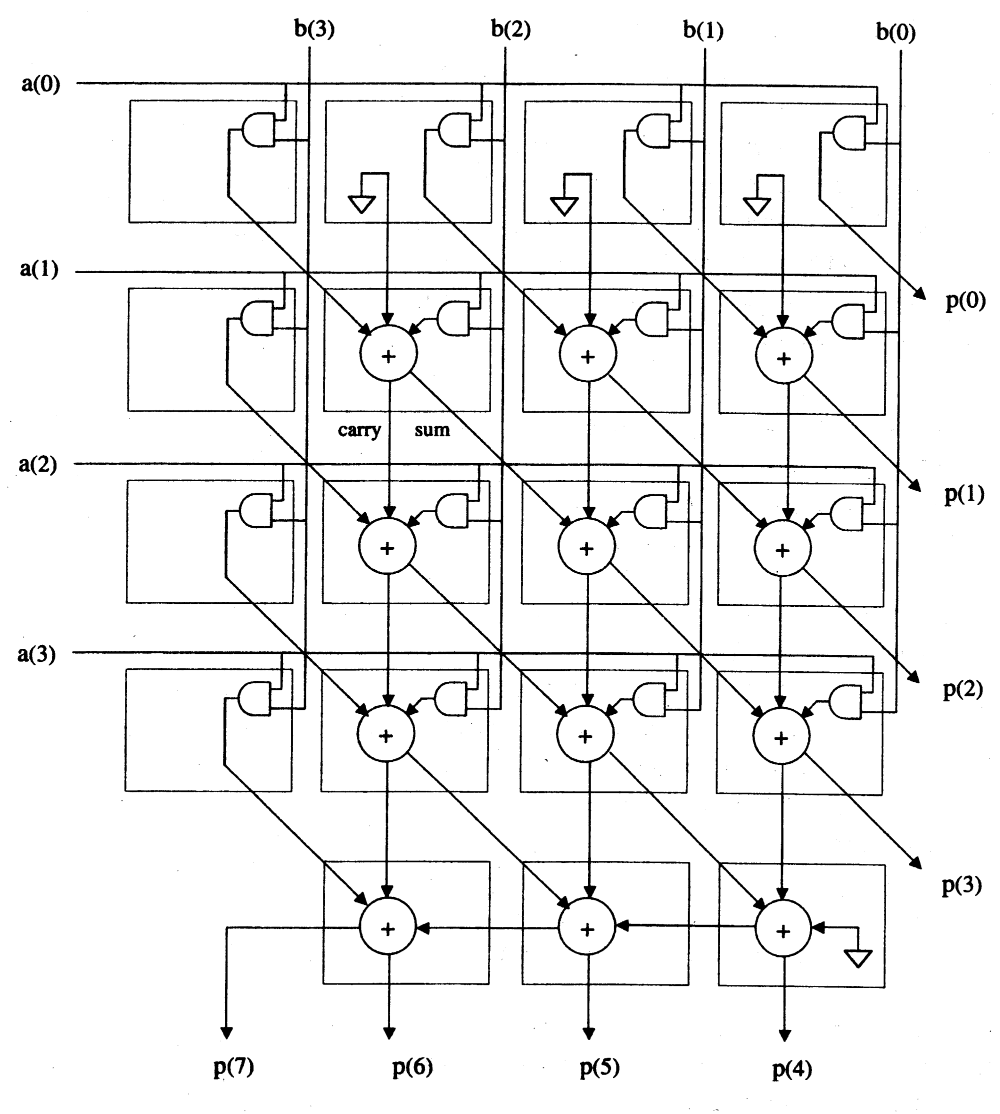


图 12.3　并行乘法器

```
1  ---------lower_row.vhd(component): --------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_components.all;
5  -------------------------------------------------
6  ENTITY lower_row IS
7     PORT (sin, cin: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
8           p: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
```

```
9  END lower_row;
10 ---------------------------------------------------
11 ARCHITECTURE structural OF lower_row IS
12    SIGNAL local: STD_LOGIC_VECTOR(2 DOWNTO 0);
13 BEGIN
14    local(0) <= '0';
15    U1: COMPONENT fau PORT MAP(sin(0), cin(0), local(0),
16       p(0), local(1));
17    U2: COMPONENT fau PORT MAP(sin(1), cin(1), local(1),
18       p(1), local(2));
19    U3: COMPONENT fau PORT MAP(sin(2), cin(2), local(2),
20       p(2), p(3));
21 END structural;
22 ---------------------------------------------------
```

```
1  --------my_components.vhd(package):--------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------------
5  PACKAGE my_components IS
6  ---------------------------------------------------
7  COMPONENT and_2 IS
8     PORT (a, b: IN STD_LOGIC; y: OUT STD_LOGIC);
9  END COMPONENT;
10 ---------------------------------------------------
11 COMPONENT fau IS          --full adder unit
12    PORT (a, b, cin: IN STD_LOGIC; s, cout: OUT STD_LOGIC);
13 END COMPONENT;
14 ---------------------------------------------------
15 COMPONENT top_row IS
16    PORT (a: IN STD_LOGIC;
17          b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
18          sout, cout: OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
19          p: OUT STD_LOGIC);
20 END COMPONENT;
21 ---------------------------------------------------
22 COMPONENT mid_row IS
23    PORT (a: IN STD_LOGIC;
24          b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
```

```
25          sin, cin: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
26          sout, cout: OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
27          p: OUT STD_LOGIC);
28 END COMPONENT;
29 -------------------------------------------------
30 COMPONENT lower_row IS
31    PORT (sin, cin: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
32          p: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
33 END COMPONENT;
34 -------------------------------------------------
35 END my_components;
36 -------------------------------------------------
```

```
1  ---------multiplier.vhd(Project): ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_components.all;
5  -------------------------------------------------
6  ENTITY multiplier IS
7     PORT (a, b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
8           prod: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9  END multiplier;
10 -------------------------------------------------
11 ARCHITECTURE structural OF multiplier IS
12    TYPE matrix IS ARRAY (0 TO 3)OF
13       STD_LOGIC_VECTOR (2 DOWNTO 0);
14    SIGNAL s, c: matrix;
15 BEGIN
16    U1: COMPONENT top_row PORT MAP (a(0), b, s(0), c(0),
17       prod(0));
18    U2: COMPONENT mid_row PORT MAP (a(1), b, s(0), c(0), s(1),
19       c(1), prod(1));
20    U3: COMPONENT mid_row PORT MAP (a(2), b, s(1), c(1), s(2),
21       c(2), prod(2));
22    U4: COMPONENT mid_row PORT MAP (a(3), b, s(2), c(2), s(3),
23       c(3), prod(3));
24    U5: COMPONENT lower_row PORT MAP (s(3), c(3),
25       prod(7 DOWNTO 4));
26 END structural;
27 -------------------------------------------------
```

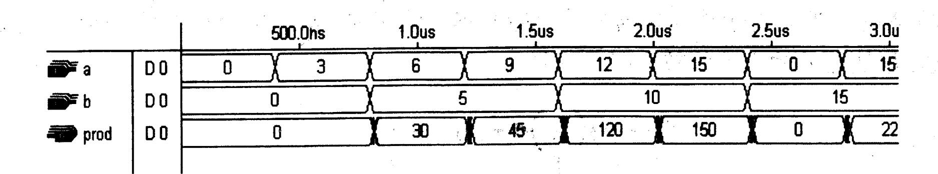
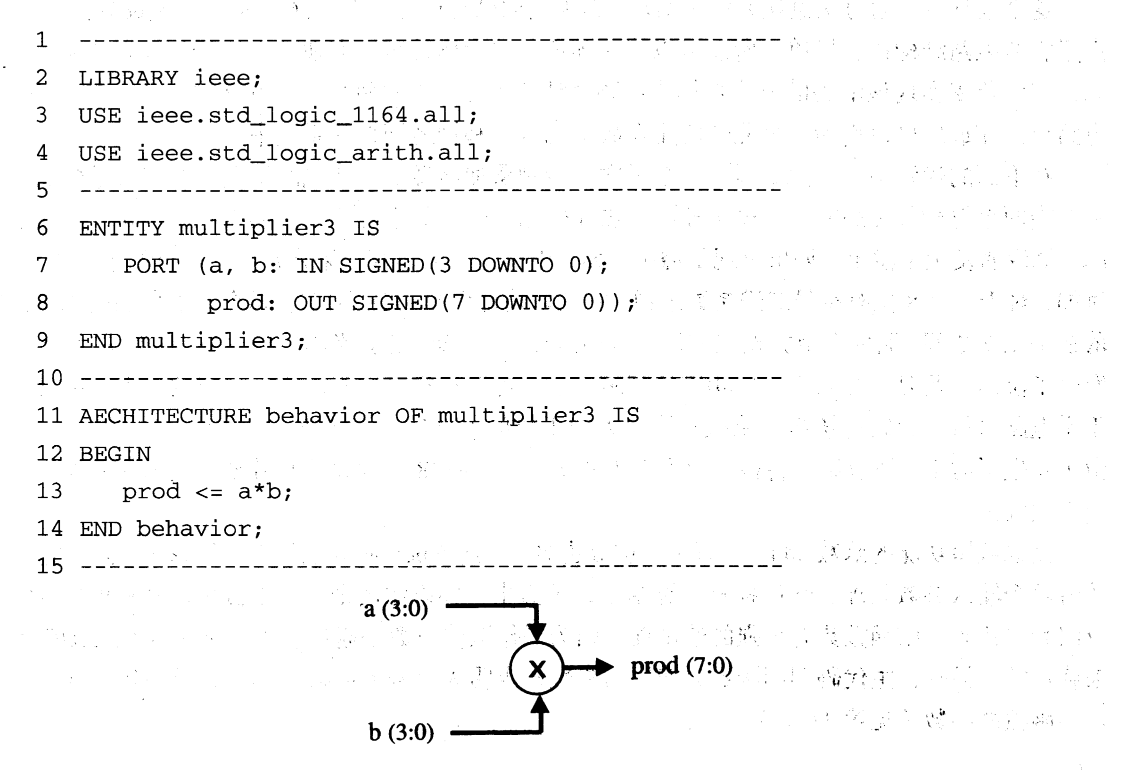


图 12.4　并行乘法器的仿真结果

更简单的乘法器实现方法

上面的两个例子是为了说明如何利用 VHDL 进行系统设计的问题。实际上在某些特殊情况下，并行乘法器可以通过对预定义的“*”操作符进行直接综合得到。因此，上面的电路可以用图 12.5 的简捷形式重新描述，上面的代码可以由下面的代码替换。



```
1  --------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;
5  --------------------------------------------------
6  ENTITY multiplier3 IS
7     PORT (a, b: IN SIGNED(3 DOWNTO 0);
8           prod: OUT SIGNED(7 DOWNTO 0));
9  END multiplier3;
10 --------------------------------------------------
11 AECHITECTURE behavior OF multiplier3 IS
12 BEGIN
13    prod <= a*b;
14 END behavior;
15 --------------------------------------------------
```


图 12.5　预定义的“*”运算符对应的电路结构

12.3　乘-累加电路

在目前的很多数字系统中，乘法器和累加器需要组合起来使用。这种乘-累加组合在那些内部高度互连的系统中经常出现，如数字滤波器、神经网络、数据量化器等。

图 12.6 给出了一个典型的乘-累加（MAC：Multiply-ACcumulate）电路结构。它首先将两个

输入的数值相乘，然后将运算结果与先前存储的累加值相加。MAC 电路的另一个特点是必须检测溢出，这在累加器的输入数值较大时很容易发生。

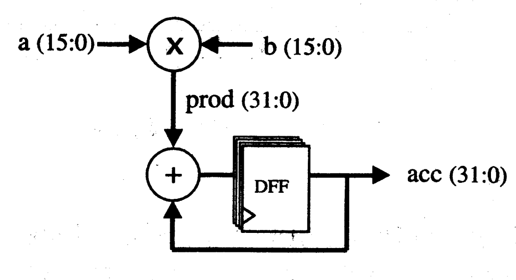


图 12.6　MAC 电路

这个设计可以通过实例化元件来实现，因为我们已经设计了图 12.6 中的每个电路单元。这里，由于它的电路结构比较简单，所以可以使用预定义的操作符直接实现其中的部分运算功能。通过元件实例化来实现该电路的方法将作为习题供读者思考，这里将使用更简捷的方法。整个 MAC 电路设计完成后可以作为一个元件应用在数字滤波器和神经网络等系统中。

在下面的代码中，专门编写了一个函数来检测累加器溢出后造成的错误。在有符号数加法器中，当两个符号位（最左边一位）相同的操作数相加时，如果计算结果的符号位与两个操作数不同，就说明发生了溢出。如果出现了溢出，那么电路返回的运算结果将是正的或负的最大边界值。例如，对于一个 8 位有符号二进制数来说，它的取值范围是–128~+127。两个正数相加的结果必须落在 1~127 之间，而两个负数相加的结果必须落在–1~–128 之间。例如，65+65 = 130，它大于 127，发生了溢出。此时，如果不采取相应的措施，对二进制有符号数来说实际返回的结果是–126。对于本电路来说，此时应该将结果截短为最大的正数值（127）。同样地，(–70) + (–70) =–140，由于发生了溢出，所以结果将被截短为最大的负数值（–128）。当两个操作数的符号位不同时，不会发生溢出。

完成截短功能的函数 add_truncate( )被放进名为 my_function 的包集中（见第 10 章）。这个函数将两个输入参数相加，然后检查计算结果是否溢出，如果溢出则将结果截短，并将处理的结果返回给主代码。该函数具有很强的通用性，因为操作数的位数是通过名为 size 的 GENERIC 参数来确定的。另外，主代码调用函数时使用的参数被声明为信号（见第 14 行），这是因为变量不能作为函数的参数（见第 11 章）。

```
1  ------PACKAGE my_functions: ---------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;
5  -------------------------------------------------
6  PACKAGE my_functions IS
7     FUNCTION add_truncate (SIGNAL a, b: SIGNED; size: INTEGER)
```

```
8         RETURN SIGNED;
9   END my_functions;
10  -------------------------------------------------
11  PACKAGE BODY my_functions IS
12     FUNCTION add_truncate (SIGNAL a, b: SIGNED; size: INTEGER)
13           RETURN SIGNED IS
14        VARIABLE result: SIGNED (size DOWNTO 0);
15   BEGIN
16        result := a+b;
17        IF (a(a'left)=b(b'left)) AND
18             (result(result'LEFT)/=a(a'left)) THEN
19          result := (result'LEFT => a(a'LEFT),
20                     OTHERS => NOT a(a'left));
21        END IF;
22        RETURN result;
23    END add_truncate;
24  END my_functions;
25  -------------------------------------------------
```

```
1   -------------Main code: ------------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   USE ieee.std_logic_arith.all;
5   USE work.my_functions.all;
6   -------------------------------------------------
7   ENTITY mac IS
8      PORT (a, b: IN SIGNED(3 DOWNTO 0);
9            clk, rst: IN STD_LOGIC;
0            acc: OUT SIGNED(7 DOWNTO 0));
11  END mac;
12  -------------------------------------------------
13  ARCHITECTURE rtl OF mac IS
14     SIGNAL prod, reg: SIGNED(7 DOWNTO 0);
15  BEGIN
16     PROCESS (rst, clk)
17        VARIABLE sum: SIGNED(7 DOWNTO 0);
18    BEGIN
19       prod <= a*b;
20       IF (rst = '1') THEN
```

```
21          reg <= (OTHERS => '0');
22        ELSIF (clk'EVENT AND clk = '1') THEN
23          sum := add_truncate (prod, reg, 8);
24          reg <= sum;
25        END IF;
26      acc <= reg;
27      END PROCESS;
28 END rtl;
29 ---------------------------------------------
```

图 12.7 给出了仿真结果。在仿真过程中 a 的取值依次为 0，2，4，6，–8，–6，–4 和–2，b 的取值依次为 0，3，6，–7，–8，–8 和–8。因此，期望的输出序列 acc 的取值应为 0，6，30，–12（在图中用 256–12 = 244 来表示–12），52，100 和 148。除了最后一个数值因超过 127 而被截短外，其他所有的值都是正确的。

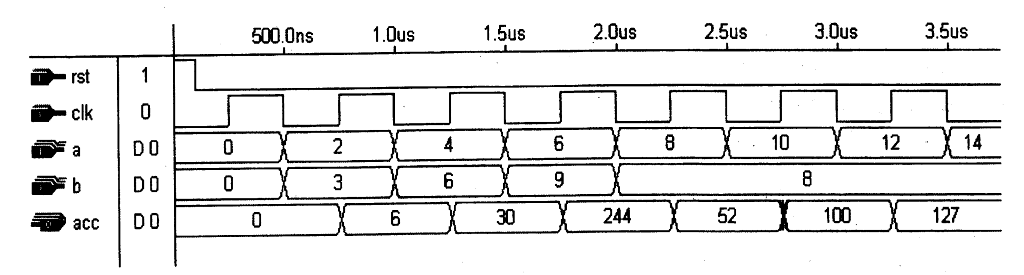


图 12.7 MAC 电路的仿真结果

12.4 数字滤波器

数字信号处理（DSP）技术在音频、视频以及通信等领域有极为广泛的应用。这些应用大多数都属于线性时不变（LTI：Linear Time Invariant）系统，LTI 系统可以采用数字电路来实现。

任何一个 LTI 系统都可以用下面的方程表示：

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

其中，a_k 和 b_k 是滤波器的系数，$x[n-k]$ 和 $y[n-k]$ 是当前（$k=0$）和以前（$k>0$）时的输入和输出值。为了实现这个表达式，需要使用寄存器来存储 $x[n-k]$ 和 $y[n-k]$（$k>0$）。除此之外还需要乘法器和加法器，这些都是在数字领域里很常用的电路单元。

数字滤波器根据其冲激响应可以分为两类：无限冲激响应（IIR：Infinite Impulse Response）滤波器和有限冲激响应（FIR：Finite Impulse Response）滤波器。IIR 是上面的方程式所描述的一般情况，而 FIR 只有当 $N = 0$ 时才可以用上面的方程式来描述。因为只有 FIR 滤波器才有线性相

位，当需要线性相位时就需要 FIR 滤波器。在通信领域中，FIR 滤波器有广泛的应用。

当 $N=0$ 时，上面的方程就变为：

$$y[n]=\sum_{k=0}^{M}c_k x[n-k]$$

其中，$c_k=b_k/a_0$ 是 FIR 滤波器的系数。这个方程可以由图 12.8 中给出的原型电路来实现，其中 D 代表寄存器，三角形代表乘法器，带有“+”的圆圈代表加法器。

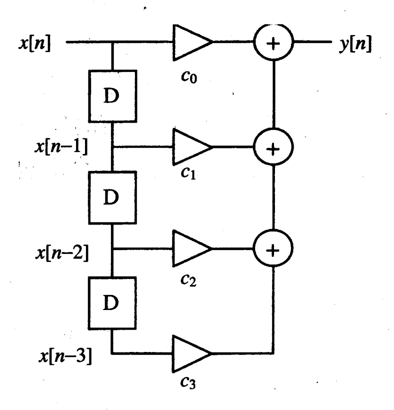


图 12.8　FIR 滤波器的原型电路

图 12.9 是实现 FIR 滤波器的实际电路结构图。如图所示，x 的值存储在移位寄存器中，寄存器的输出与乘法器相连，乘法器的输出连接到加法器上。FIR 滤波器的系数同样也要存储在片上。如果系数都是固定的（例如专用滤波器），它们的值可以通过逻辑门来存储，而不需要使用寄存器（因为此时仅需要存储常量）。如果它是通用的滤波器，就需要寄存器来存储系数，以便于根据具体需要进行修改。在图 12.9 中，为了使输出波形稳定，没有毛刺，采用的是同步输出方式。

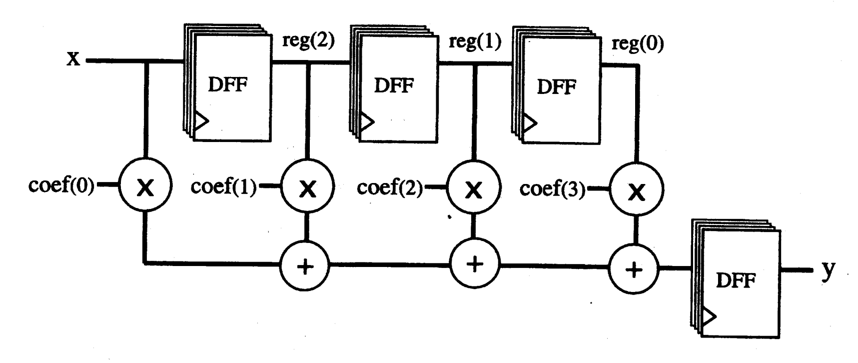


图 12.9　实际 FIR 滤波器的电路结构

图 12.9 所示电路的具体实现方式可以有很多种。然而，为了实现代码的重用与共享，应该尽可能使用 GENERIC 参数。在下面的代码中使用了两个 GENERIC 参数（见第 7 行），n 定义为滤

波器系数的个数，m 是系数和输入信号的位数。输出信号的宽度是 2m 位。假如，x，coef 和 reg 都是 16 位的，那么 y 和与之相连的其他信号都是 32 位的。

观察图 12.9 可以发现，它的下部是一个由 MAC 构成的流水线结构，与 12.3 节分析的 MAC 电路十分相似。同样可以想像，在这里也会发生运算溢出，所以在设计中必须包括对累加结果的截短操作。

在下面的代码中，系数是常量（第 19 行），因此不会占用寄存器资源。取 coef(0) = 4，coef(1) = 3，coef(2) = 2 和 coef(3) = 1，其中 m 和 n 取值较小，目的是使仿真结果更直观。因为 m = n = 4，所以整个电路综合后共需要 20 个触发器（每级移位寄存器需要 4 个，加上 8 个输出寄存器，共需要 20 个）。根据第 7 章所学的内容，如果一个信号的赋值是通过其他信号的跳变来触发的，那么电路综合后就会生成寄存器（见第 33 行~第 45 行）。下面的代码中的第 33 行~第 38 行是对变量进行赋值的语句，由于这些变量的值被赋给信号 y，所以综合后也会生成寄存器。

```
1  --------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;      -- SIGNED类型需要的包集
5  --------------------------------------------------
6  ENTITY fir2 IS
7     GENERIC (n: INTEGER := 4; m: INTEGER := 4);
8     --n=# of coef., m=# of bits of input and coef.
9     --Besides n and m, CONSTANT (line 19) also need adjust
10    PORT (x; IN SIGNED(m-1 DOWNTO 0);
11            clk, rst: IN STD_LOGIC;
12            y: OUT SIGNED(2*m-1 DOWNTO 0));
13 END fir2;
14 --------------------------------------------------
15 ARCHITECTURE rtl OF fir2 IS
16    TYPE registers IS ARRAY (n-2 DOWNTO 0) OF
17                                 SIGNED(m-1 DOWNTO 0);
18    TYPE coefficients IS ARRAY (n-1 DOWNTO 0) OF
19                                     SIGNED (m-1 DOWNTO 0);
20    SIGNAL reg: registers;

21    CONSTANT coef: coefficients := ("0001", "0010", "0011",
22                                        "0100");
23 BEGIN
24    PROCESS (clk, rst)
25       VARIABLE acc, prod:
```

```
26             SIGNED(2*m-1 DOWNTO 0) := (OTHERS => '0');
27          VARIABLE sign: STD_LOGIC;
28    BEGIN
29       ---------------reset: ----------------------
30       IF (rst = '1') THEN
31          FOR i IN n-2 DOWNTO 0 LOOP
32             FOR j IN m-1 DOWNTO 0 LOOP
33                reg(i)(j) <= '0';
34             END LOOP;
35          END LOOP;
36       ----------register inference + MAC: ---------
37       ELSIF (clk'EVENT AND clk = '1') THEN
38          acc := coef(0)*x;
39          FOR i IN 1 TO n-1 LOOP
40             sign := acc(2*m-1);
41             prod := coef(i)*reg(n-1-i);
42             acc := acc+prod;
43          -------overflow check: ------------------
44             IF (sign=prod(prod'left))AND
45                  (acc(acc'left)/=sign)
46                THEN
47                acc := (acc'LEFT => sign , OTHERS =>  NOT sign);
48             END IF;
49          END LOOP;
50          reg <= x&reg(n-2 DOWNTO 1);
51       END IF;
52       y <= acc;
53    END PROCESS;
54 END rtl;
55 --------------------------------------------------
```

仿真结果如图 12.10 所示。FIR 滤波器的系数分别为 coef(0) = 4，coef(1) = 3，coef(2) = 2 和 coef(3) = 1，需要注意的是系数都是有符号的（因此对于 4 位值的范围为–8~7）。输入序列 x 的取值分别为 x[0] = 0，x[1] = 5，x[2] = –6（图中为 16–6 = 10），x[3] = –1（图中为 16–1 = 15），x[4] = 4，x[5] = –7（图中为 16–7 = 9）和 x[6] = –2（图中为 16–2 = 14）。在所有触发器已经复位的情况下，根据前面的公式，可以对输出结果的期望值进行计算：

```
y[0]=coef(0)*x(0)=0
y[1]= coef(0)*x[1]+ coef(1)*x[0]=20
```

```
y[2]= coef(0)*x[2]+ coef(1)*x[1]+ coef(2)*x[0]=-9
......
```

这与图 12.10 给出的结果是相同的。

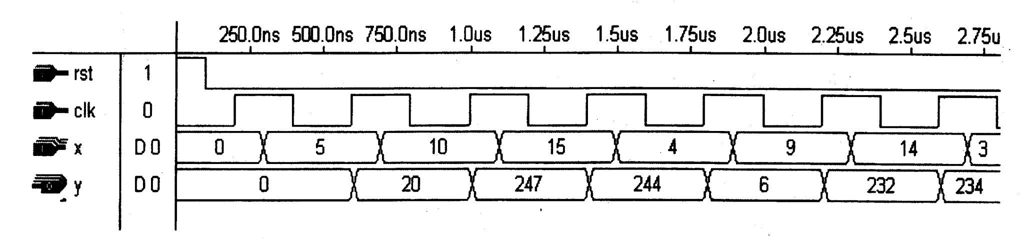


图 12.10　FIR 滤波器的仿真结果

通用的 FIR 滤波器

上面将系数固定的做法适用于设计专用滤波器。要实现通用的 FIR 滤波器（即系数可以根据需要改变），可以采用图 12.11 所示的电路结构。这个电路结构是模块化的，通过几个相同模块的级联，可以满足不同的设计要求。这在实际应用中是非常重要的，因为不同的应用场合所需的 FIR 滤波器的级数也是不同的。

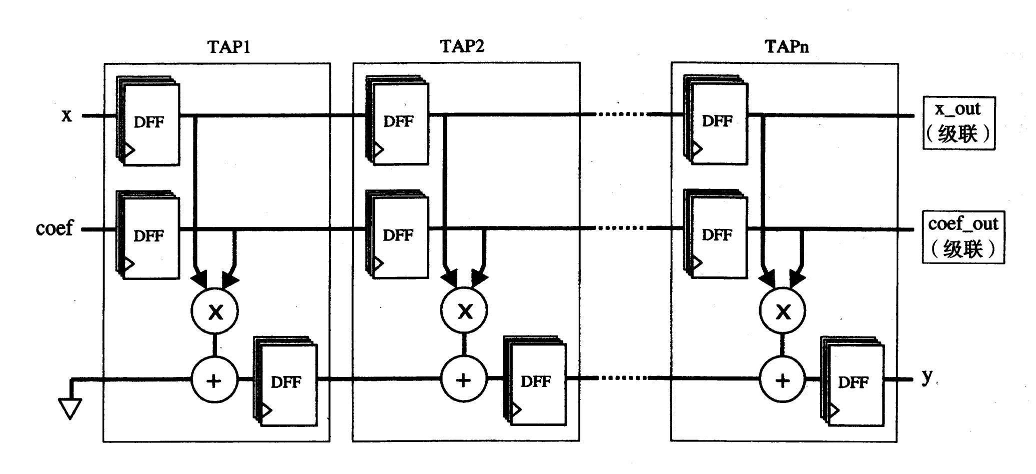


图 12.11　通用 FIR 滤波器

图 12.11 所示的电路由 n 个模块（TAP）级联而成。每个 TAP 都包括两组移位寄存器，一组用来存储输入信号（x），另一组用来存储系数（coef）。此外，每个 TAP 中还包括一个乘法器、一个加法器和一组可选的输出寄存器。如果在每个 TAP 中都使用输出寄存器，将会增加相邻两个加法器之间的延迟。当然，所有的系数都要在电路开始进行计算之前完成加载。这种 FIR 滤波器结构的设计将作为习题供读者思考。

12.5　神经网络

神经网络（NN：Neural Networks）是高度并行、高度互连的系统。这使得它的实现富有挑战性，同时需要较高昂的代价，因为需要占用大量的硬件资源。

图 12.12（a）给出了前馈神经网络的电路结构示意图。在图中，电路被划分为 3 层，每层有 3 个 3 输入的神经元。每层内部的具体连接关系见图 12.12（b）。图中，x*i* 代表第 *i* 个输入，w*ij* 代表输入 *i* 和神经元 *j* 之间的权值（weight），y*j* 是第 *j* 个输出。如图 12.12（b）所示，可以得到：

```
y1=f(x1.w11+x2.w21+x3.w31)
y2=f(x1.w12+x2.w22+x3.w32)
y3=f(x1.w13+x2.w23+x3.w33)
```

其中 f( )是激化函数（如线性阈值的 sigmoid 函数等）。

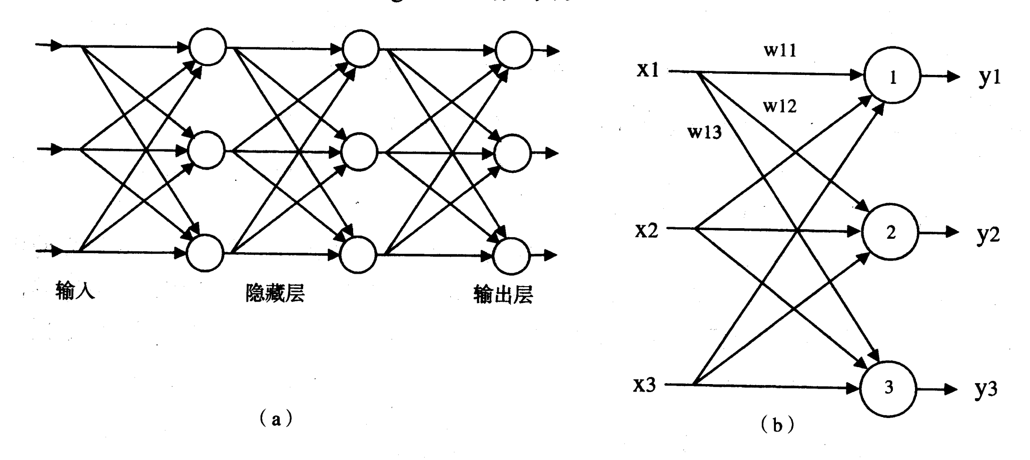


图 12.12　前馈神经网络

图 12.13 所示的环形结构是为实现图 12.12 中的前向神经网络而设计的，它可以实现图 12.12(b)的功能。图中每个矩形框内部的电路构成一个神经元。如图 12.12 所示，图中在垂直方向上有几组环行移位寄存器，在水平方向上有一个大的环形结构。垂直的环行移位寄存器中存储着前面所提到的权值，水平环行移位寄存器中装载的是输入信号。每个权值在自己的移位寄存器中的相对位置必须和输入值匹配。在每个垂直环形移位寄存器的输出端有一个 MAC 电路（见 12.3 节），MAC 用来对权值和输入信号进行乘-累加运算。所有的移位寄存器都使用相同的工作时钟。因此，在一个完整的循环后，MAC 电路的输出结果分别为：x1*w11+x2*w21+x3*w31，x1*w12+x2*w22+x3*w32 和 x1*w13+x2*w23+x3*w33。这些运算结果送给查找表（LUT：Look Up Table），用于实现激化函数，最终产生 NN 所需的输出结果 yi。

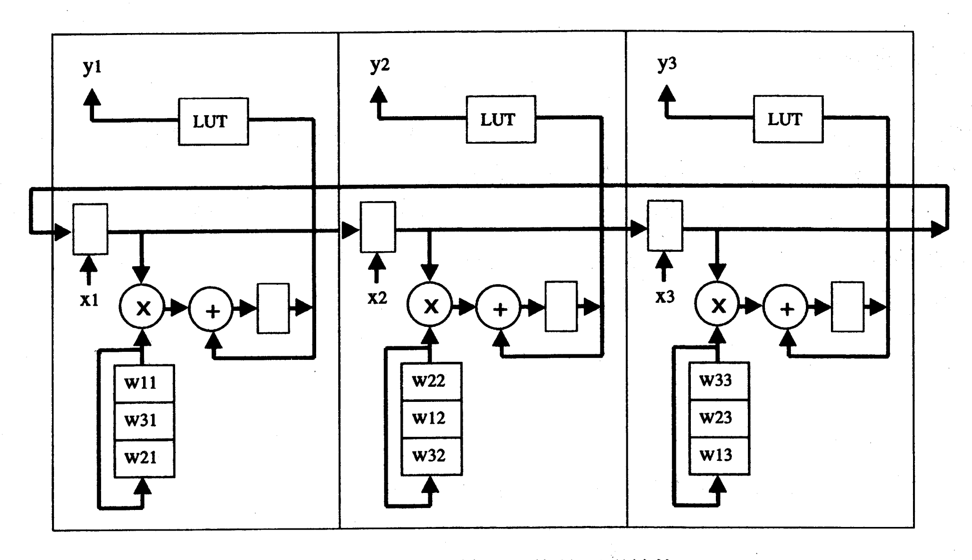


图 12.13　实现神经网络的环形结构

在这种电路中，必须考虑累加结果的截短问题。如果输入信号和权值都是 16 位长的，那么 MAC 的输出很自然地选取 32 位宽度。但由于 *yi* 可能需要作为其他神经元的输入，所以需要截短到 16 位，这可以在 LUT 或 MAC 中实现。

另一种实现 NN 的方法见图 12.14，适合于实现权值可编程的通用神经网络。只需要使用一个输入端口来加载所有的权值（可以节省芯片的引脚数量）。在图 12.14 中，权值按照顺序移位，直到每个寄存器都存储相应的权值。然后权值与输入相乘并累加，最终得到期望的输出结果。

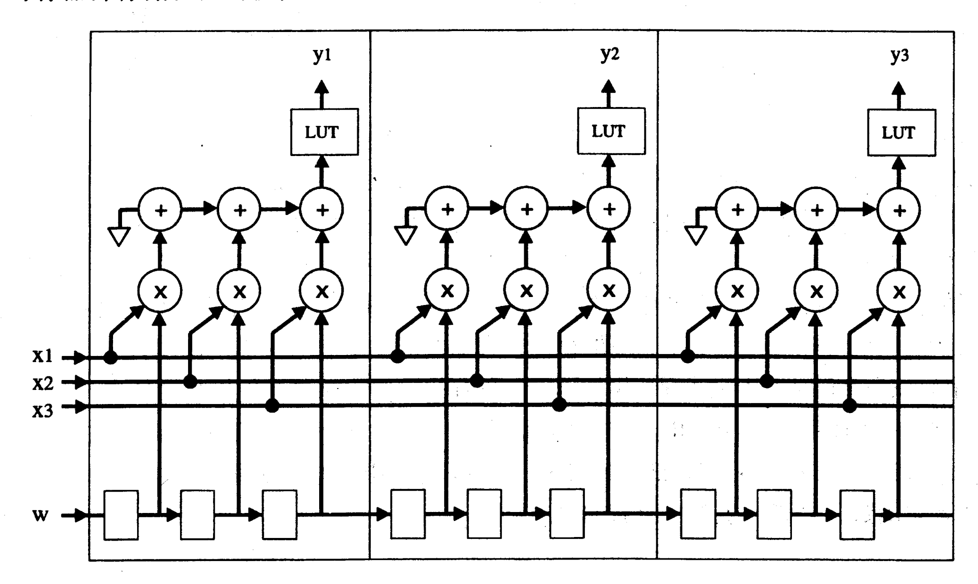


图 12.14　只有一个权值输入端的神经网络的电路结构

下面给出了两段 VHDL 代码，都实现了图 12.14 给出的电路。然而两种方法都没有使用 LUT（这将在习题 12.5 中处理）。这两种方法的主要区别在于，第一段代码不是通用的，因此适合参数确定的小型设计，而第二种方法是通用的，可以重用而且很容易调整神经网络的大小。

方法一：小型神经网络的设计

这个方法的主要优点是简单，容易理解，采用一段主代码就可以实现。惟一的局限就是它的输出和输入都需要逐个单独列出，而不是使用二维数组，因此不适合实现较大的神经网络。

```
1  ---------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;   -- SIGNED 类型需要的包集
5  ---------------------------------------------------
6  ENTITY nn IS
7  GENERIC (n: INTEGER := 3;                  -- 神经元的数量
8           m: INTEGER := 3;                  -- 每个神经元输入信号与权值的个数
9           b: INTEGER := 4);                 -- 每个输入信号或权值的位宽
10 PORT (x1: IN SIGNED(b-1 DOWNTO 0);
11       x2: IN SIGNED(b-1 DOWNTO 0);
12       x3: IN SIGNED(b-1 DOWNTO 0);
13       w: IN SIGNED(b-1 DOWNTO 0);
14       clk: IN STD_LOGIC;
15       test: OUT SIGNED(b-1 DOWNTO 0);    -- 寄存器测试输出
16       y1: OUT SIGNED(2*b-1 DOWNTO 0);
17       y2: OUT SIGNED(2*b-1 DOWNTO 0);
18       y3: OUT SIGNED(2*b-1 DOWNTO 0));
19 END nn;
20 ---------------------------------------------------
21 ARCHITECTURE neural OF nn IS
22    TYPE weights IS ARRAY (1 TO n*m) OF SIGNED(b-1 DOWNTO 0);
23    TYPE inputs IS ARRAY (1 TO m) OF SIGNED(b-1 DOWNTO 0);
24    TYPE outputs IS ARRAY (1 TO m) OF SIGNED(2*b-1 DOWNTO 0);
25 BEGIN
26    PROCESS (clk, w, x1, x2, x3)
27       VARIABLE weight: weights;
28       VARIABLE input: inputs;
29       VARIABLE output: outputs;
30       VARIABLE prod, acc: SIGNED(2*b-1 DOWNTO 0);
31       VARIABLE sign: STD_LOGIC;
```

```
32    BEGIN
33 -----------shift register inference: -------------
34       IF (clk'EVENT AND clk = '1') THEN
35          weight := w&weight(1 TO n*m-1);
36       END IF;
37 ----------------inictialization-------------------
38       input(1) := x1;
39       input(2) := x2;
40       input(3) := x3;
41 ---------------multiply-accumulate: --------------
42       L1: FOR i in 1 TO n LOOP
43         acc := (OTHERS => '0');
44       L2: FOR j IN 1 TO m LOOP
45         prod := input(j)*weigth (m*(i-1)+j);
46         sign := acc(acc'LEFT);
47         acc := acc+prod;
48 -----------------overflow check: -----------------
49         IF (sign=prod(prod'left))AND
50               (acc(acc'left)/=sign) THEN
51            acc := (acc'LEFT => sign, OTHERS => NOT sign);
52         END IF;
53       END LOOP L2;
54       output(i) := acc;
55    END LOOP L1;
56 ----------outputs: -------------------------------
57    test <= weight(n*m);
58    y1 <= output(1);
59    y2 <= output(2);
60    y3 <= output(3);
61   END PROCESS;
62 END neural;
63 -------------------------------------------------
```

图 12.15 给出了电路的仿真结果。为了使仿真结果更直观，这里采用的输入/输出位数和神经元数量都比较少。从代码的第 7 行~第 9 行可以看出，这个神经网络有 3 个神经元，每个神经元有 3 个输入信号，输入信号以及权值的位宽都为 4，并且都是有符号的，因此输入信号和权值的取值范围是−8~7。输出值的范围是−128~127。在仿真时，输入信号可以固定为 x1 = 3，x2 = 4 和 x3 = 5。因为共有 9 个权值，所以需要 9 个时钟周期来将它们移入，如图 12.15 所示。在仿真时，9 个权值

分别是 w9 = 1，w8 = 2，…，w1 = 9（注意，第一个输入的权值是 w9，它要进行 9 次移位）。因为使用的是有符号数类型的信号，所以 9 实际上是–7，8 实际上是–8。当权值全部装载完成以后，系统立即计算它的第一组输出：

```
y1= x1.w1+x2.w2+x3.w3=(3)(-7)+(4)(-8)+(5)(7)=-18        （表示为 256-18=238）
y2= x1.w4+x2.w5+x3.w6=(3)(6)+(4)(5)+(5)(4)=58
y3=x1.w7+x2.w8+x3.w9=(3)(3)+(4)(2)+(5)(1)=22
```

这些值可以在图 12.15 的右侧看到。

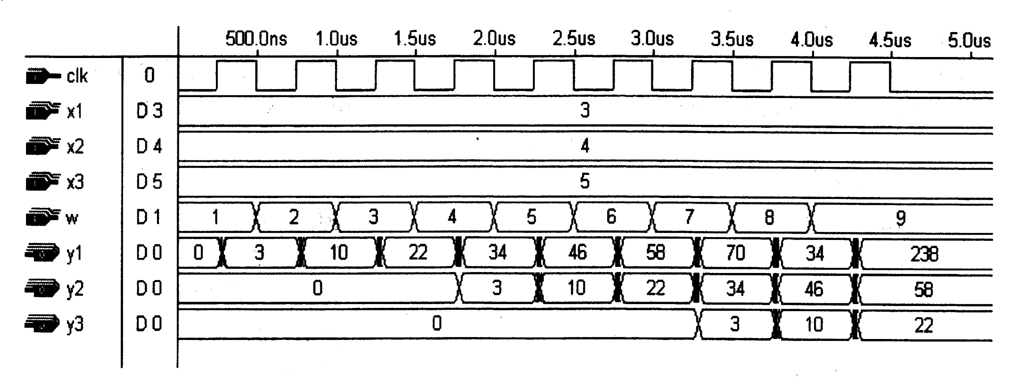


图 12.15　第一种神经网络实现方法的仿真结果

方法二：大规模神经网络的实现

下面的代码具有很强的通用性。其输入/输出是用二维数组（见 3.5 节）来描述的，因此可以方便地改变神经网络的大小。

为了定义设计中需要的输入和输出数组，这里使用了一个名为 my_data_types 的包集。该包集中包含两个用户定义的数据类型 vector_array_in 和 vector_array_out。使用该包集需要在主代码中加入相关的 USE 语句。这些数据类型用来定义电路的输入和输出（分别在第 13 行和第 18 行中定义）。由于所有的参数现在都是通用的，容易调整，因此可以构建不同规模的神经网络。

下面的代码被划分为两个部分：第 28 行~第 32 行的顺序逻辑部分（实现移位寄存器）和紧随其后的组合逻辑部分（实现 MAC）。这里有一个可选的测试输出信号，用于对最后一组寄存器的值进行检测。正如先前设计的 MAC 电路，这里必须进行溢出检测和处理（见第 42 行~第 45 行）。

```
1  --------------Package my_data_types:--------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;            -- SIGNED 类型需要的包集
5  -------------------------------------------------
6  PACKAGE my_data_types IS
7     CONSTANT b: INTEGER := 3;             -- 输入位数或权重
```

```
8     TYPE vector_array_in IS ARRAY(NATURAL RANGE<>)OF
9                                        SIGNED(b-1 DOWNTO 0);
10    TYPE vector_array_out IS ARRAY(NATURAL RANGE<>)OF
11                                        SIGNED(2*b-1 DOWNTO 0);
12 END my_data_types;
13 --------------------------------------------------
```

```
1  ------Project nn (main code): -------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;              -- SIGNED类型需要的包集
5  USE work.my_data_types.all;                -- 用户定义类型的包集
6  --------------------------------------------------
7  ENTITY nn3 IS
8     GENERIC (n: INTEGER := 3;               -- 神经元数
9              m: INTEGER := 3;               -- 输入的数目或
10                                            -- 每神经元的权重
11             b: INTEGER := 3);              -- 每输入的位数
12                                            -- 或加权
13     PORT (x: IN VECTOR_ARRAY_IN(1 TO m);
14           w: IN SIGNED(b-1 DOWNTO 0);
15           clk: IN STD_LOGIC;
16           test: OUT SIGNED(b-1 DOWNTO 0); -- 寄存器
17                                            -- 测试输出
18           y: OUT VECTOR_ARRAY_OUT(1 TO n));
19 END nn3;
20 --------------------------------------------------
21 ARCHITECTURE neural OF nn3 IS
22 BEGIN
23    PROCESS (clk, w, x)
24       VARIABLE weight: VECTOR_ARRAY_IN(1 TO m*n);
25       VARIABLE prod, acc: SIGNED(2*b-1 DOWNTO 0);
26       VARIBALE sign: STD_LOGIC;
27    BEGIN
28 --------shift register inference: ---------------
29       IF (clk'EVENT AND clk = '1')THEN
30          weight: = w&weight(1 TO n*m-1);
31       END IF;
32       test <= weight(n*m);
```

```
33 ------------initialization: ----------------------
34       acc := (OTHERS => '0');
35 -------------multiply-accumulate: ----------------
36      L1: FOR i IN 1 TOn LOOP
37        L2: FOR j IN 1 TO m LOOP
38          prod:= x(j)*weight(m*(i-1)+j);
39          sign := acc(acc'LEFT);
40          acc := acc+prod;
41 -----------------overflow check: -----------------
42          IF (sign = prod(prod'LEFT))AND
43                (acc(acc'LEFT)/=sign) THEN
44             acc := (acc'LEFT => sign, OTHERS => NOT sign);
45          END IF;
46       END LOOP L2;
47 -------------output: ----------------------------
48       y(i) <= acc;
49       acc := (OTHERS => '0');
50     END LOOP L1;
51   END PROCESS;
52 END neural;
53 -------------------------------------------------
```

其他与神经网络有关的内容将在习题 12.5 中进行分析。

12.6　习题

下列习题的重点是练习使用包集、元件、函数和过程进行系统级电路的设计。

12.1　并行乘法器。 在 12.2 节中可以看到实现一个并行乘法器的完整过程，其中提到使用预定义的乘法运算符“*”也可以实现并行乘法器。虽然有很多种电路结构可以实现乘法运算功能（图 12.3 介绍了其中的一种），但有理由认为 12.2 节中给出的两种设计方法（从底层开始设计的方法和使用乘法运算操作符的方法）所消耗的硬件资源不会有悬殊的差别。综合这两种实现方法对应的代码，并比较最后的报告文件。选择几种不同类型的 PLD/FPGA 为目标芯片，观察每种情况下需要占用多少硬件资源，并比较它们是否在同一量级上。

12.2　移位寄存器。 图 P12.2 是一个 4 级移位寄存器的电路结构图。输出信号（q）是通过多路复用器进行选择的，其中数据总线宽度为 8（所以每一级寄存器包括 8 个 D 触发器）。现在需要进行以下工作：

（a）创建两个元件 reg 和 mux，然后利用元件构建完整的如图 P12.2 所示的电路。

（b）假定仅仅实现移位寄存器而不需要多路复用器，并且所有寄存器的输出都作为输出端

口。编写这样一个电路的 VHDL 代码。

(c)设计与(b)类似的电路，但移位寄存器的级数(n)和每一级的宽度(b)都是 GENERIC 参数。在这种情况下，输出信号（qout）应采用用户自定义的数组类型。建议回顾 3.5 节或者分析 12.5 节中的第二个设计。

最后，继续上面的步骤(c)，要在移位寄存器中增加数据加载功能。给每个寄存器增加一个额外的输入（称为 x）和一个额外的 load 引脚。当 load 有效时，用 x 输入的值来覆盖所有寄存器的当前值。对于 x，可以具有与 qout 相同的自定义数据类型。

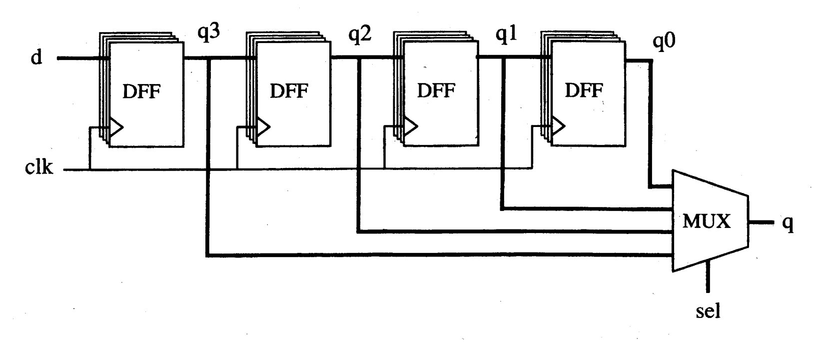


图 P12.2

12.3 **MAC 电路**。在 12.3 节中使用函数设计了 MAC 电路（见图 12.6）。在本习题中，要求使用元件（乘法器、加法器和寄存器）来实现该电路。创建所需元件并在主代码中对它们进行实例化。对设计进行编译和仿真，将仿真结果与图 12.7 的结果进行比较。

12.4 **通用 FIR 滤波器**。在 12.4 节中讨论了 FIR 滤波器的实现，给出了一个完整的设计实例，其中滤波器的参数是固定的。为了实现通用 FIR 滤波器，图 12.11 建议使用模块化的结构。编写代码实现通用 FIR 滤波器（见 12.3 节和 12.4 节）。在设计中加入溢出检测。从输入信号（x 和 coef）到乘法器的输入信号的位宽都为 m，从乘法器输出到 y 的位宽均为 2m。这里的级数选择为 n。要求代码的通用性较强。最后对设计进行综合与仿真。

12.5 在 12.5 节中讨论了神经网络这种高度互连系统的实现，并给出了两种电路结构以及针对第二种结构的两段 VHDL 代码，然而这些代码中都没有包含 LUT（查找表）。在这个习题中，要求完成以下工作：

(a)编写 VHDL 代码实现 LUT。LUT 实际上是一种简单的 ROM，可参考在 9.10 节中 ROM 的实现方法。

(b)编写代码实现图 12.13 所描述的神经网络结构，综合和仿真该设计并验证其正确性。

(c)除了 12.5 节介绍的两种方法以外，还有其他实现神经网络的方法，能举出另一种方法吗？能对现有方法提出改进意见吗？

附录 A　可编程逻辑器件

A.1　简介

可编程逻辑器件（PLD）的概念于 20 世纪 70 年代中期提出，目的在于构造可编程组合逻辑电路。与利用固定硬件结构运行软件程序的微处理器不同，PLD 的可编程性是在硬件逻辑层次上的。换言之，PLD 是一种通用芯片，它内部的硬件资源可以根据特定的需求进行重新配置。

最早的可编程逻辑器件根据其物理结构和编程方式被称为可编程阵列逻辑（PAL：Programmable Array Logic）或可编程逻辑阵列（PLA：Programmable Logic Array）。由于没有使用触发器而只包括逻辑门，所以只能应用于组合逻辑电路的设计中。后来为了克服这个问题，在电路的每个输出端都附加了一个寄存器，这类器件称为 registered PLD，使可编程逻辑器件也能够应用于简单时序电路中。

20 世纪 80 年代初，PLD 的输出端又增加了附加的逻辑电路。新的输出单元被称为宏单元，除寄存器以外，它还包含逻辑门和多路复用器。宏单元本身也是可编程的，具有多种操作模式，并且还提供从电路输出到可编程逻辑阵列内部的一个反馈信号，从而使可编程逻辑器件具有更大的灵活性。这种新的可编程逻辑器件结构被称为通用 PAL（即 GAL）。此外还有一种采用类似结构的器件称为 PALCE（CMOS 电可擦写/可编程 PAL）。

PAL，PLA 和 Registered PLD 以及 GAL/PALCE 统称为简单可编程逻辑器件。随后人们将多个 GAL 芯片集成在一起，采用了更复杂的编程技术和更先进的硅工艺，并增加了对边界扫描测试技术（JTAG）的支持，增加了接口逻辑电平种类，这样的器件称为复杂可编程逻辑器件（CPLD）。CPLD 具有集成度高、性能好、成本低等特点，从而应用得越来越广泛。

20 世纪 80 年代中期，FPGA（现场可编程门阵列）出现了。虽然 FPGA 在结构、技术、工作特点、成本等方面都不同于 CPLD，但两者都可用于大规模高性能电路的设计。

下表是简要的 PLD 发展过程：

<table>
<tr><td rowspan="3">可编程逻辑器件（PLD）</td><td>简单可编程逻辑器件（SPLD）</td><td>可编程逻辑阵列（PLA）
可编程阵列逻辑（PAL）
寄存器可编程阵列逻辑（Registered PAL）/可编程逻辑阵列（PLA）
通用阵列逻辑（GAL）</td></tr>
<tr><td colspan="2">复杂可编程逻辑器件（CPLD）</td></tr>
<tr><td colspan="2">现场可编程门阵列（FPGA）</td></tr>
</table>

还需要说明的一点是，所有的 PLD（无论是简单的还是复杂的）都是非易失性的。采用熔丝或反熔丝工艺的器件只能进行一次性编程，而采用 EEPROM 或 Flash ROM（Flash 可应用于很多新型器件的制造）的可编程逻辑器件则是可以重新编程的。另一方面，由于使用 SRAM 来存储芯片的连接关系等编程信息，所以 FPGA 多数是易失性的，这时就需要使用外部 ROM，用于上电后对 FPGA 进行配置。当然，在采用熔丝工艺后 FPGA 也可以具有非易失性。

A.2 简单可编程逻辑器件

PAL，PLA 和 GAL 统称为简单可编程逻辑器件（SPLD），下面将对它们所采用的各种结构进行介绍。

可编程阵列逻辑器件（PAL）

PAL 结构于 20 世纪 70 年代中期由 Monolithic Memories 公司提出。它的基本结构如图 A.1 所示，图中的空心圆代表可编程的连接点。从图中可以看出，该电路由连接关系可编程的与门阵列加上固定的或门阵列组成。

图中所示结构的实现基于“所有的组合逻辑功能都相当于一个乘积项之和”这一基本结论。也就是说，如果 a_1，a_2，⋯，a_N 是逻辑输入，而任意组合输出 x 可以表示为：

$$x = m_1 + m_2 + \cdots + m_M$$

其中，$m_i=f_i(a_1, a_2, \cdots, a_N)$是 x 的部分项。例如，

$$x = a_1\overline{a_2} + a_2a_3\overline{a_4} + \overline{a_1a_2}a_3a_4\overline{a_5}$$

由于其中的乘积项可以通过与门来实现，然后在输出端使用或门来计算它们的和，所以可以实现上述等式的功能。

这种处理方法的局限性在于只能实现组合逻辑功能。为了解决这个问题，在 20 世纪 70 年代末提出了 Registered PAL 的概念。该芯片在每个输出端（在图 A.1 中的或门的后面）加入了一个寄存器，从而可以实现一些简单时序功能。

当时流行的一种 PAL 芯片是 PAL16L8，它有 16 个输入端和 8 个输出端（除电源和地以外，该芯片共有 18 个有效 I/O 引脚，其中 10 个为输入引脚，2 个为输出引脚，6 个为双向的输入/输出引脚）。与 PAL16L8 对应的含有寄存器的芯片是 PAL16R8（R 表示寄存器）。

早期的 PAL 采用双极型工艺，用 5 V 电源供电，并有 200 mA 的电流消耗，其最高工作频率约为 100 MHz，采用的编程技术是 PROM（熔丝连接）或 EPROM（通过大于 20 分钟的紫外线照射后可以对以前的配置进行擦除）。

可编程逻辑阵列器件（PLA）

PLA 器件结构于 20 世纪 70 年代中期提出，基本结构如图 A.2 所示。对比图 A.1 可以发现，PLA 的与门阵列和或门阵列都是可编程的，这显然具有更高的灵活性。但更多的内部节点级数会带来更大的时延，从而降低了芯片的运行速度。

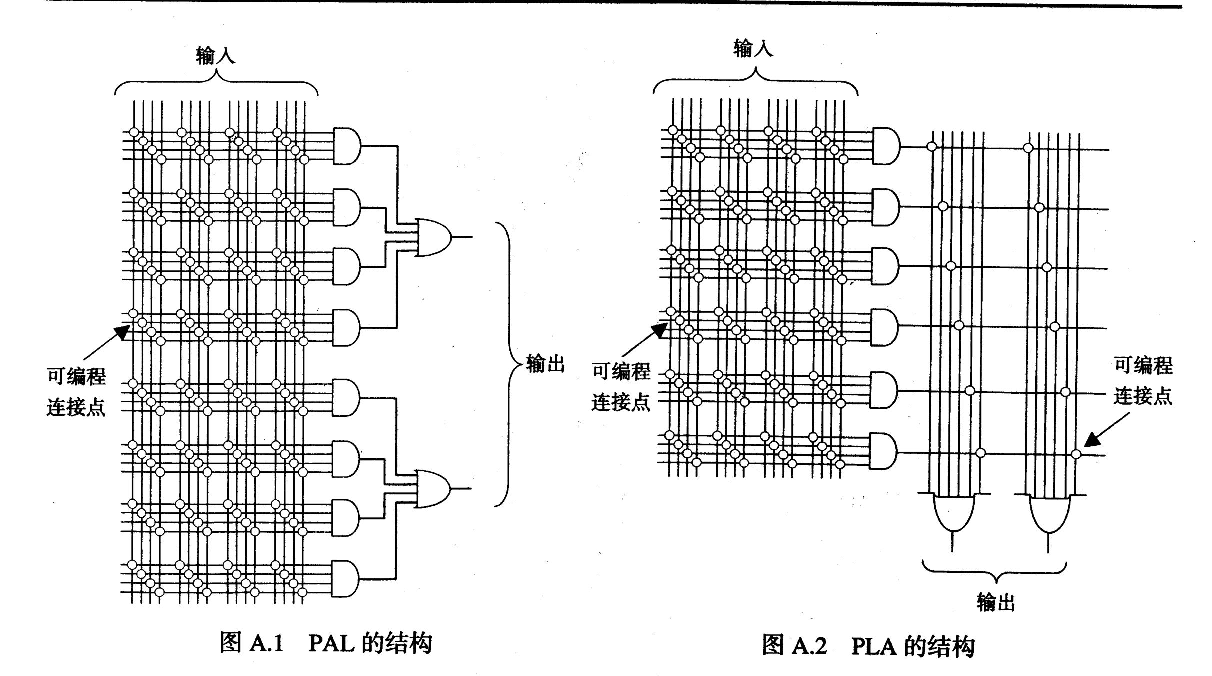


图 A.1　PAL 的结构　　　图 A.2　PLA 的结构

当时流行的一种 PLA 器件是美国 Signetics 公司的 PLS161，它的内部包括 48 个 12 输入的与门和 8 个 48 输入的或门，在输出端还有附加的异或门供编程使用。

PLA 采用与 PAL 相同的制造工艺。尽管 PLA 现在已经被淘汰了，但近来作为基本单元又被应用于 Xilinx 公司的 CoolRunner 系列低功耗 CPLD 中，这也是目前的第一个低功耗 CPLD 产品系列。

通用可编程阵列逻辑器件（GAL）

GAL 的结构于 20 世纪 80 年代初由 Lattice 公司提出，相比第一代可编程阵列逻辑器件有一些重要的改进。首先，构造了更复杂的输出单元——宏单元，该单元除寄存器外还包含有一些门电路和多路复用器；第二，宏单元本身是可编程的，提供多种运算模式；第三，宏单元可以向可编程阵列提供一个反馈信号，从而使该电路可以实现更复杂的功能；第四，以 EEPROM 代替了 PROM/EPROM，从而可以包含电子签名认证功能。

正如前面所提到的，GAL 仍然是独立封装制造的简单可编程逻辑器件。另外，GAL 作为基本组合单元被应用于大多数 CPLD 中（也有例外，如前面提到的 CoolRunner 系列 CPLD 中采用的就是 PLA）。

图 A.3 显示了 GAL16V8 的结构（V 代表多功能），整个芯片包括 20 个引脚，最多可以配置 16 个输入引脚，8 个输出引脚。如图所示，第 2 引脚到第 9 引脚是 8 个输入引脚，第 12 引脚到第 19 引脚是双向输入/输出引脚，第 1 引脚是附加时钟引脚（CLK），第 11 引脚是输出使能引脚（/OE），第 20 引脚是电源引脚（VDD），第 10 引脚是接地引脚（GND）。在每个输出端都含有一个带寄存器、逻辑门和多路复用器的宏单元（Macrocell），在图中可以看到从宏单元到可编程阵列的反馈信号。图中的空心圆代表可编程连接，读者可以发现，除输出宏单元和反馈信号以外，其他结构和图 A.1 中的 PAL 的结构类似。

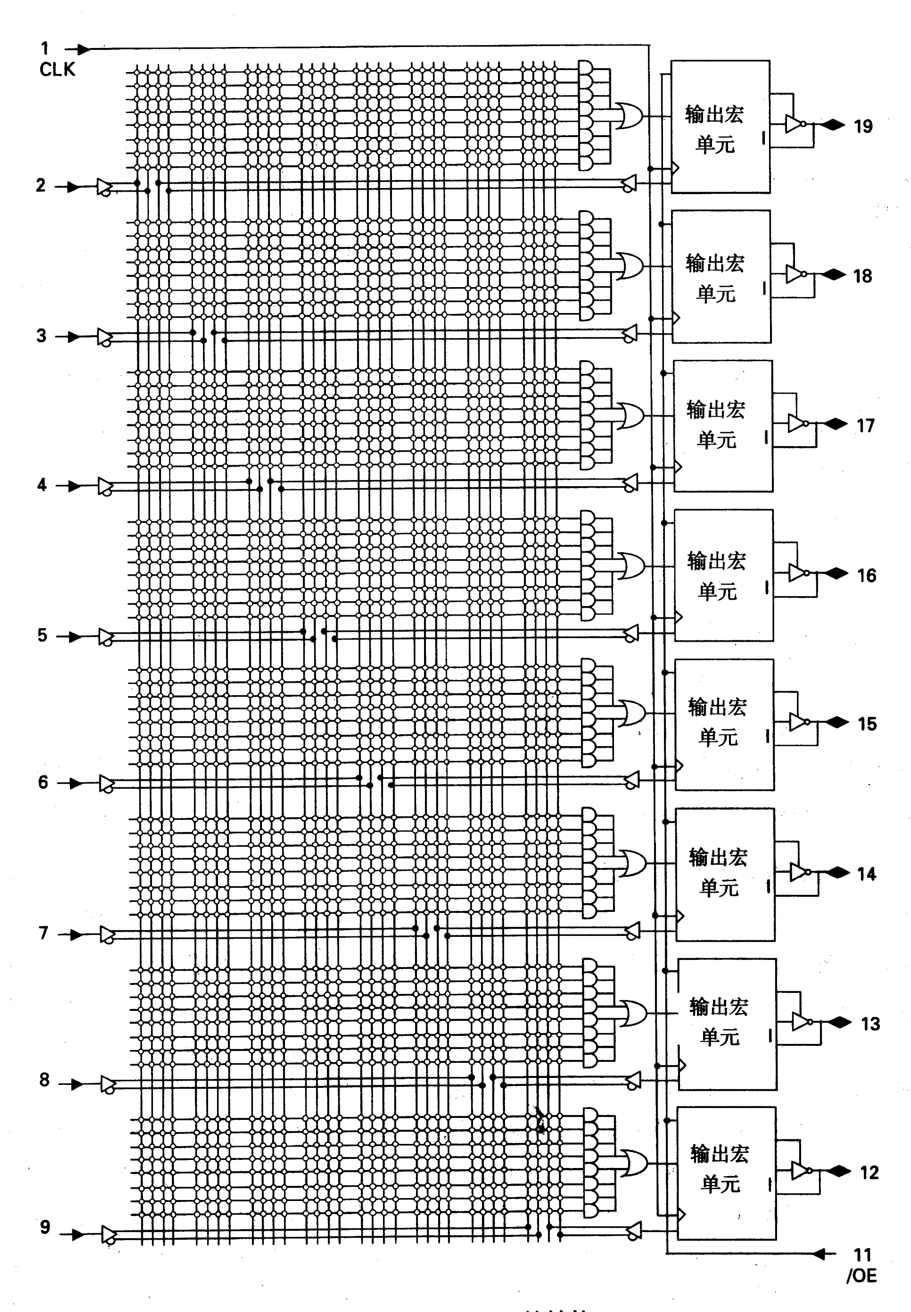


图 A.3　GAL16V8 的结构

现在的 GAL 器件采用 CMOS 工艺，3.3 V 电源，EEPROM 或 Flash 工艺，其最高工作频率为 250 MHz，主要制造公司有 Lattice，Atmel 和 TI 等。

A.3　复杂可编程逻辑器件（CPLD）

CPLD 的基本结构如图 A.4 所示，由若干个通常为 GAL 类型的 PLD 集成在单一芯片上，以及一个将其连接到 I/O 引脚的可编程开关阵列共同构成。此外，CPLD 还具有一些附加特性，如支持 JTAG 和目前常用的 I/O 接口标准。

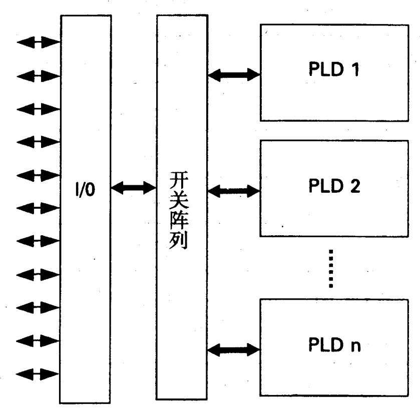


图 A.4　CPLD 的结构

在图 A.4 中，以 Xilinx 公司的 XC9500 系列 CPLD 为例。它含有 n 个类似 GAL36V18 的器件（与图 A.3 中的 GAL16V8 结构类似，但并不是 16 个输入端和 8 个输出端，而是 36 个输入端和 18 个输出端，并带有 18 个宏单元），其中 n = 2，4，6，8，12 或 16。

Altera，Xilinx，Lattice，Atmel 和 Cypress 等公司都有自己的 CPLD 产品。表 A.1 和表 A.2 给出了 Altera 和 Xilinx 公司的部分典型产品。可以发现，这些器件中可以包含数百个宏单元和上万个逻辑门。

表 A.1　Altera 的 CPLD

系列	Max7000（B，AE，S）	MAX3000（A）	MAXⅡ（G）
宏单元/查找表（LUT）	32~512 个宏单元	32~512 个宏单元	240~2210 个查找表（192~1700 个等效宏单元）
系统门	600~10 000	600~10 000	
I/O 引脚	32~512	34~208	80~272
最高内部频率	303 MHz	227 MHz	304 MHz（I/O 限制）
支持电压	2.5 V（B），3.3 V（AE）和 5V（S）	3.3 V	1.8 V（G），2.5 V，3.3 V
编程方式	EEPROM	EEPROM	Flash+SRAM
静态电流	9~450 mA	9~150 mA	2~50 mA
工艺	0.22 μm CMOS EEPROM 4 层金属（7000B）	0.3 μm, 4 层金属	0.18 μm, 6 层金属

A.4 现场可编程门阵列（FPGA）

FPGA 器件于 20 世纪 80 年代中期由 Xilinx 公司提出，在结构、存储工艺、芯片规模和成本上与 CPLD 不同，其目标在于实现高性能的大规模电路。

表 A.2 Xilinx 的 CPLD

系列	XC9500（XV，XL，−）	CoolRunner X PLA3	CoolRunner Ⅱ
宏单元	36~288	32~512	32~512
系统门	800~6400	750~12 000	750~12 000
I/O 引脚	34~192	36~260	33~270
最高内部频率	222 MHz	213 MHz	385 MHz
组成结构	GAL 54V18（XV，XL） GAL 36V18（−）	PLA 模块	PLA 模块
支持电压	2.5 V（XV），3.3 V（XL），5 V	3.3 V	1.8 V
编程方式	Flash	EEPROM	
工艺	0.35 μm CMOS	0.35 μm CMOS	0.18 μm CMOS
静态电流	11~500 mA	<0.1 mA	22 μA~1mA

图 A.5 显示了 FPGA 的基本结构，包括多个可配置逻辑模块（CLB：Configurable Logic Blocks），之间通过多个开关阵列进行互连。

CLB（见图 A.5）的内部结构与 PLD（见图 A.4）不同。首先，CLB 是基于查找表（LUT：Look Up Table）结构的。此外，FPGA 使用了比 CPLD 更多的寄存器，从而可以用于设计更复杂的时序电路。除了提供对 JTAG 的支持和对不同逻辑接口标准的支持以外，FPGA 还有很多额外特性，例如内部集成了 SRAM 存储器、倍频器（锁相环和数字锁相环）、支持 PCI 接口电平规范等，部分芯片还集成了乘法器、DSP 和微处理器等功能强大的内核。

FPGA 与 CPLD 的另一个本质区别是其配置信息的存储方式。CPLD 是非易失性的（即采用熔丝工艺、EEPROM 和 Flash 等），而 FPGA 多使用 SRAM，从而是易失性的。由于 FPGA 提供了大量可编程连接，却只需要一个外部 ROM（用于存储 FPGA 的配置信息），从而节省了空间并降低了成本。另外，当无需进行多次编程时，采用反熔丝工艺的非易失的 FPGA 会更有效。

目前 FPGA 的设计技术十分先进。现在最新的 FPGA 采用 0.09 μm，9 层铜互连的 CMOS 工艺，有上千个 I/O 引脚。图 A.6 中是目前一些 FPGA 采用的封装形式，图中左边的封装有 64 个引脚，中间的封装有 324 个引脚，右边的封装有多达 1152 个引脚。

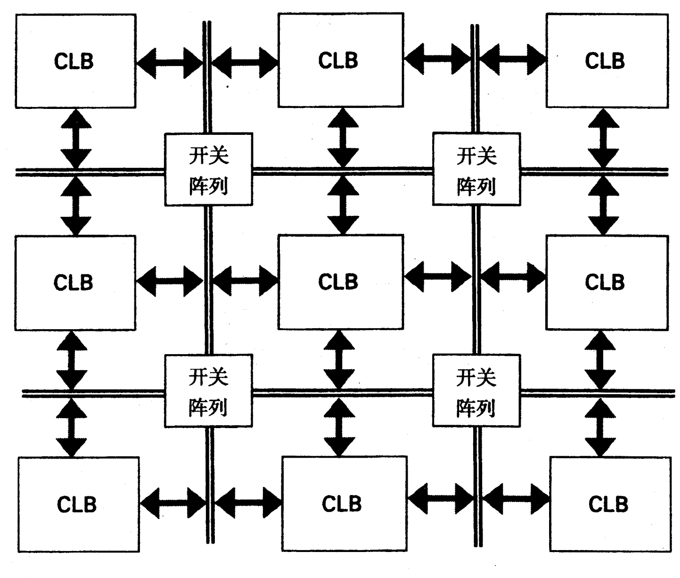


图 A.5 FPGA 的结构

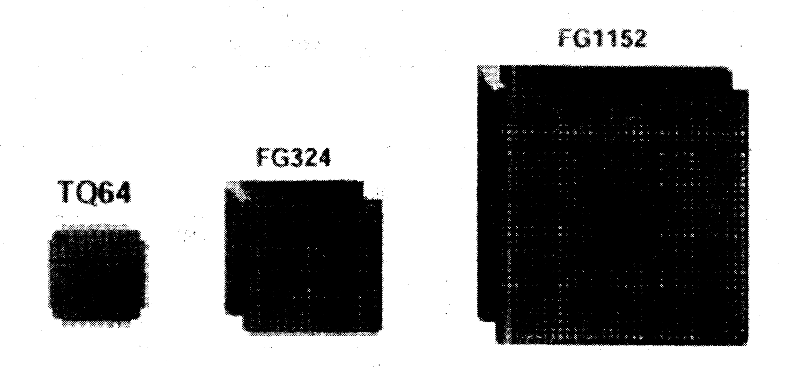


图 A.6 FPGA 封装

Xilinx，Actel，Altera，QuickLogic 和 Atmel 等公司都生产 FPGA 产品。表 A.3 和表 A.4 给出了 Xilinx 和 Actel 公司的一些产品的参数。可以发现，这些器件中可以包含上千个触发器和几百万个等效的逻辑门。

Xilinx 公司的所有 FPGA 都采用了 SRAM 来存储配置信息，从而可以实现再次编程，同时也是易失性的（需要外部 ROM）。Actel 公司的 FPGA 是非易失性的（使用反熔丝技术），但不能实现再次编程（采用 Flash 存储器的系列产品除外）。由于每个实现方法都有其优缺点，哪种芯片更适用将取决于具体的应用情况。

表 A.3 Xilinx 的 FPGA

系列	VirtexIIPro（X）	Virtex II	Virtex E	Virtex	Spartan3	Spartan IIE	Spartan II
逻辑块（CLB）	352~11 024	64~11 648	384~16 224	384~6144	192~8320	384~3456	96~1176
逻辑单元	3168~125 136	576~104 882	1728~73 008	1728~27 648	1728~74 880	1728~15 552	432~5292
系统门		40 K~8 M	72 K~4 M	58 K~1.1 M	50 K~5 M	23 ~600 K	15~200 K
I/O 引脚	204~1200	88~1108	176~804	180~512	124~784	182~514	86~284
触发器	2816~88 192	512~93 184	1392~64 896	1392 ~24 576	1536~66 560	1536~13 824	384~4704
最高内部频率（MHz）	547	420	240	200	326	200	200
支持电压（V）	1.5	1.5	1.8	2.5	1.2	1.8	2.5
编程方式	SRAM	SRAM	SRAM	SRAM	SRAM	SRAM	SRAM
工艺	0.13 μm 9 层铜 CMOS 工艺	0.15 μm 8 层金属 CMOS 工艺	0.18 μm 5 层金属 CMOS 工艺	0.22 μm 5 层金属 CMOS 工艺	0.09 μm 8 层金属 CMOS 工艺		
SRAM 位数（块 RAM）	216 K~8 M	72 K~3 M	64~832 K	32~128 K	72 K~1.8 M	32~288 K	16~56 K

表 A.4 Actel 的 FPGA

系列	Accelerator	ProASIC	MX	SX	eX
逻辑模块数	2016~32 256	5376~56 320	295~2438	768~6036	192~768
系统门	125 K~2 M	75 K~1 M	3~54 K	12~108 K	3~12 K
I/O 引脚	168~684	204~712	57~202	130~360	84~132
寄存器	1344~21 504	5376~26 880	147~1822	512~4024	128~512
最高内部频率 (MHz)	500	250	250	350	350
支持电压(V)	1.5	2.5，3.3	3.3，5	2.5，3.3，5	2.5，3.3，5
连接	反熔丝	闪存	反熔丝	反熔丝	反熔丝
工艺	0.15 μm 7 层金属 CMOS 工艺	0.22 μm 4 层金属 CMOS 工艺	0.45 μm 3 层金属 CMOS 工艺	0.22 μm CMOS 工艺	0.22 μm CMOS 工艺
SRAM 位数	29~339 K	14~198 K	2.56 K	n.a.	n.a.

附录 B　Xilinx ISE 和 ModelSim 使用指南

在本附录中将介绍以下综合、布线和仿真工具。

工具	应用	对应附录
ISE 6.1 + ModelSim 5.7c	Xilinx CPLD/FPGA	附录 B
MaxPlus II 10.2 + Advanced Synthesis Software	Altera CPLD/FPGA（部分）	附录 C
Quartus II 3.0	Altera CPLD/FPGA	附录 D

Xilinx ISE 6.1 是针对 Xilinx 公司的可编程器件产品的一整套开发环境。ModelSim XE 5.7c（Model Technology 公司的产品）是该开发环境所调用的仿真工具。

可以从 www.xilinx.com 免费下载 Xilinx ISE 6.1 和 ModelSim XE II 5.7c 的入门版本。

本附录是一个简明指南，包括与该开发环境相关的 5 个部分：

- VHDL 代码输入
- 综合和实现
- 产生测试激励
- 使用 ModelSim 进行仿真
- 物理实现

B.1　VHDL 代码输入

- 启动 ISE 6.1 工程管理器（Project Navigator），显示图 B.1 所示的界面。
- 新建一个工程(File→New Project)，显示图 B.2 所示的对话框，在 Project Name 中输入 VHDL 代码中实体的名称（例如 flipflop），在 Project Location 中选择工作目录，最后选择 HDL 作为顶层电路模块的描述语言，然后单击 Next。
- 在图 B.3 所示的对话框中选择器件和器件系列（如 Spartan 3），然后选择 XST 作为综合工具，选择 ModelSim 作为仿真工具，选择 VHDL 作为设计语言，然后单击 Next。
- 在图 B.4 所示的对话框中，选择 VHDL Module，输入文件名（例如 flipflop.vhd），选择路径，然后单击 Next，则会出现图 B.5 所示的文本编辑器。
- 输入 VHDL 代码（见图 B.5）并保存，现在工程就可以综合了。

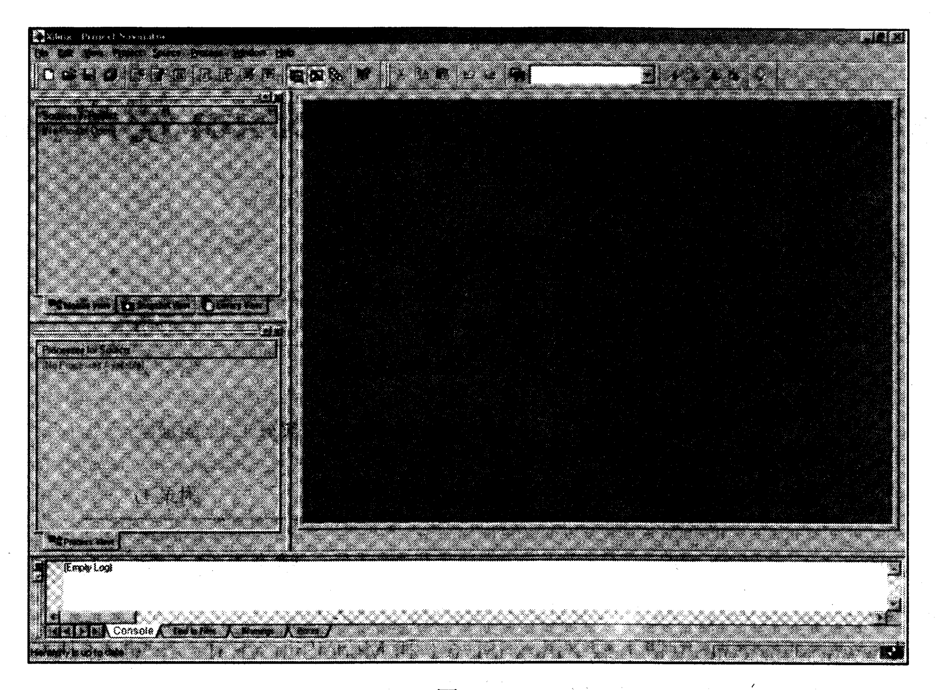

Console

图 B.1

图 B.2

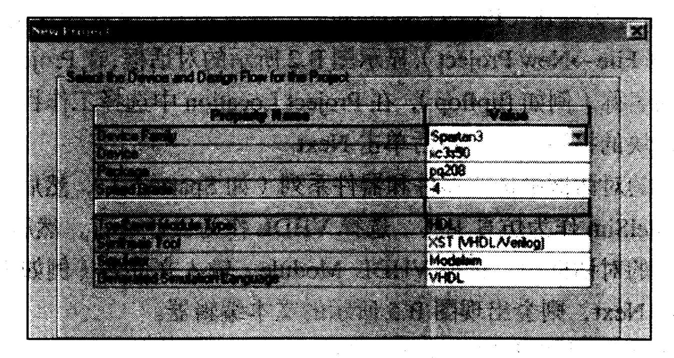

New Project
Select the Device and Design Flow for the Project
Property Name
Value
Device Family
Spartan3
Device
Package
pq208
Speed Grade
-4
Top-Level Module Type
HDL
Synthesis Tool
XST (VHDL/Verilog)
Simulator
Modelsim
Generated Simulation Language
VHDL

图 B.3

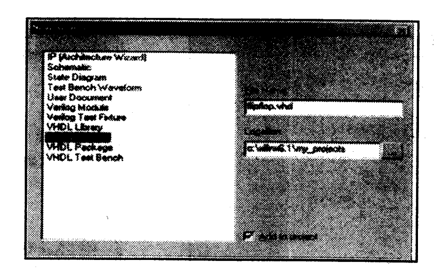


图 B.4

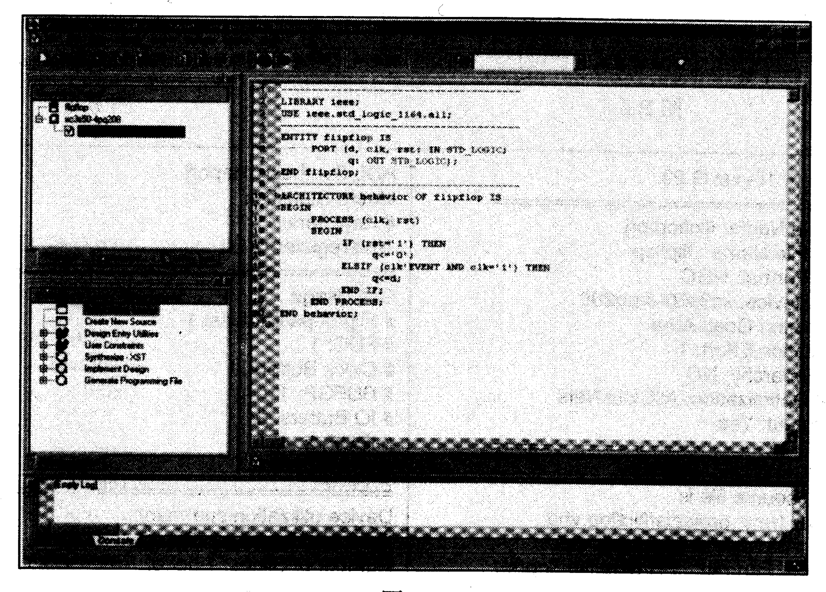


图 B.5

B.2　综合和实现

- 在处理流程窗口（Processes for Source window），选择 Synthesize-XST，然后选择 Process→Properties，在出现的图 B.6 所示的窗口中，选择综合目标为面积优先（Optimization Goal = Area），选择综合力度为普通（Optimization Effort=Normal），然后单击 OK。
- 选择 Process→Run 进行综合，也可以单击 或者双击 Synthesize-XST 进行综合。另外，如果需要，可以在综合前进行语法检验，只需单击 Synthesize-XST 前的"+"打开下拉菜单（见图 B.7），然后双击 Check Syntax 即可。
- 综合后可以通过双击 Processes for Source window 中 Synthesize-XST 下的 View Synthesis Report 查看综合报告。为能更好地查看该报告，可以使用绑定工具图标 。图 B.8 显示了这种报告的部分内容，通过报告文件可以查看编译器使用的寄存器数目等内容。
- 通过双击 Synthesize-XST 目录下的 View RTL Schematic 可以查看与代码对应的电路结构图，如图 B.9 所示。

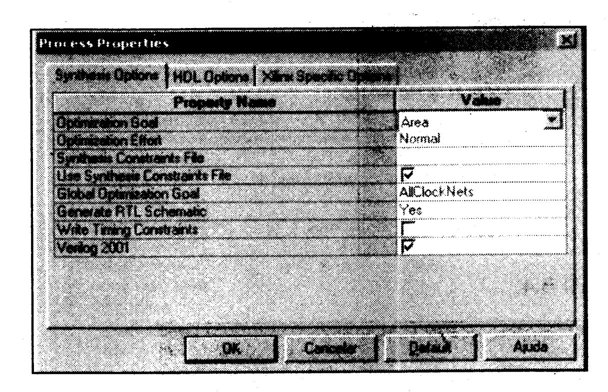


图 B.6

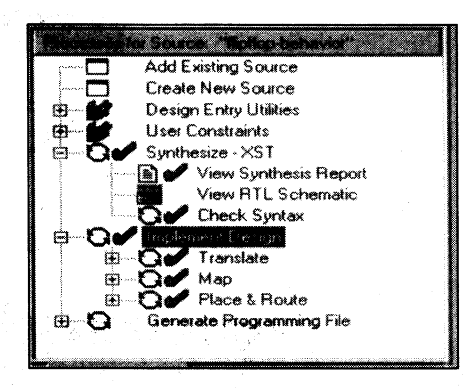
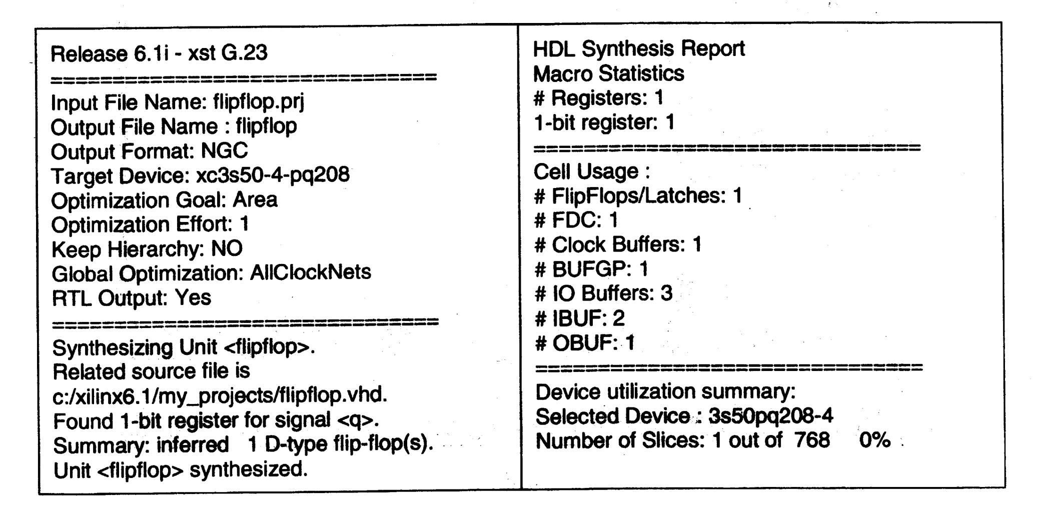


图 B.7



Release 6.1i - xst G.23
==============================
Input File Name: flipflop.prj
Output File Name : flipflop
Output Format: NGC
Target Device: xc3s50-4-pq208
Optimization Goal: Area
Optimization Effort: 1
Keep Hierarchy: NO
Global Optimization: AllClockNets
RTL Output: Yes
==============================
Synthesizing Unit <flipflop>.
Related source file is
c:/xilinx6.1/my_projects/flipflop.vhd.
Found 1-bit register for signal <q>.
Summary: inferred 1 D-type flip-flop(s).
Unit <flipflop> synthesized.

HDL Synthesis Report
Macro Statistics
Registers: 1
1-bit register: 1
==============================
Cell Usage :
FlipFlops/Latches: 1
FDC: 1
Clock Buffers: 1
BUFGP: 1
IO Buffers: 3
IBUF: 2
OBUF: 1
==============================
Device utilization summary:
Selected Device : 3s50pq208-4
Number of Slices: 1 out of 768 0%

图 B.8

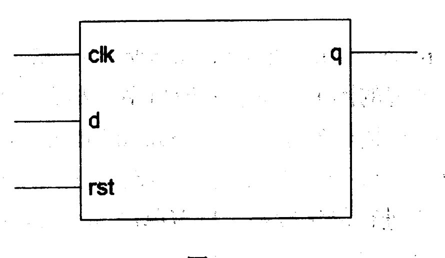


图 B.9

- 通过双击处理流程窗口（Processes for Source window）下的 Implement Design 选项实现设计（见图 B.7）。
- 设计实现后，单击 Implement Design 选项，可以检查生成的各种报告，尤其是引脚分配报告（Pad Report，在 Place & Route directory 目录下），检查信号和引脚之间的对应关系。

- 使用布局规划器（Floorplanner）。在 Place & Route 目录下双击 View/Edit Placed Design（Floorplanner），选择 View→Hierarchy，View→Floorplan，View→Placement 和 View→Package Pins，对应地都会打开一个窗口。将光标移到芯片的每个引脚上，以查看其属性。

注意，如果选用 CPLD（如 CoolRunner 系列）来代替 FPGA（如 Spartan 3 系列），处理流程窗口（Processes for Source window）的选项会有所不同。例如，双击工程窗口（Project Window）中的器件说明，将进入图 B.3 所示的对话框。试将器件改为 CoolRunner 2，单击 OK 后查看处理流程窗口（Processes for Source window）中的选项有什么不同。

B.3　产生测试激励（使用 HDL 测试文件自动生成器 HDL Bencher）

HDL Bencher 用来生成激励（或测试波形），然后调用 ModelSim 进行仿真。

- 选择 Project→New Source，就会出现图 B.10 所示的对话框，选择 Test Bench Waveform，输入文件名（例如 flipflop_tbw），检查工程路径是否正确，最后单击 Next，直至启动 HDL Bencher（见图 B.11）。

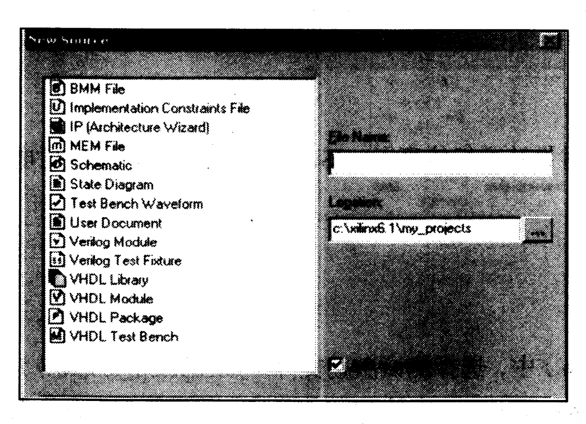


图 B.10

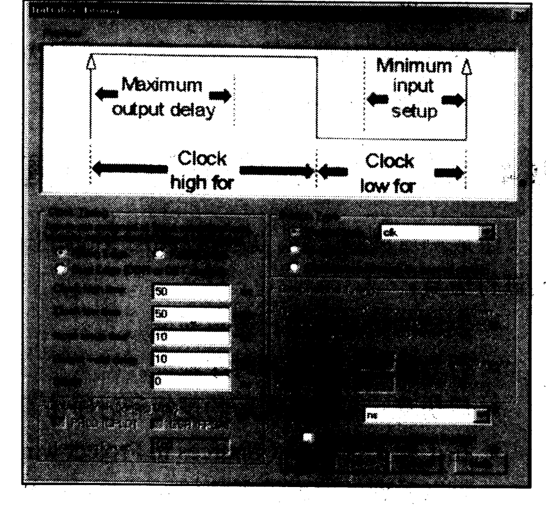


图 B.11

- 当 HDL Bencher 启动后，会显示如图 B.11 所示的界面，可以在这里设定时钟信号。输入时钟信号（clk）被设为主时钟，输入其设置参数并单击 OK，就会显示图 B.12 所示的波形窗口。
- 图 B.12 所示波形信号的位置都可以通过拖动来改变。另外，如果需要改变时钟波形，则要单击图形 ⎍ 或在波形处单击右键，这样会出现图 B.11 所示的界面，然后就可以根据需要进行修改了。

- 如果要设定其他信号的值（见图 B.12），可通过单击信号后面的纵向网格线，选择需要设定的区域和相关的电平值。设定所有输入信号的值之后，将出现图 B.13 所示的界面。

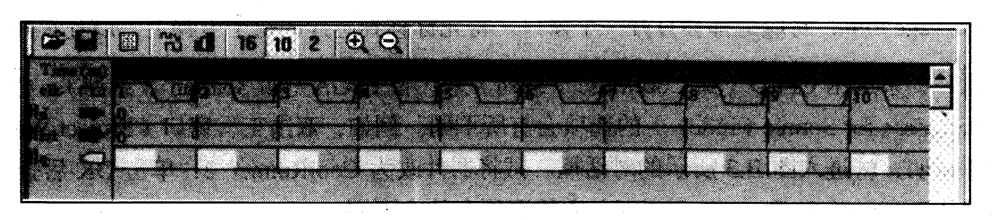


图 B.12

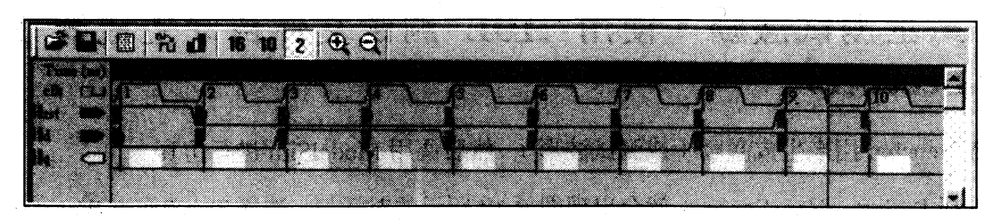


图 B.13

- 定义测试结束时间。在曲线下面的区域单击右键并选择 End of Testbench，拖动蓝线到所需位置即可。
- 保存测试激励文件。可以看到一个名为 flipflop_tbw.tbw 的新文件被加进工程窗口（Sources in Project window）。

B.4 使用 ModelSim 进行仿真

生成测试激励之后，就可以调用 ModelSim 对设计进行仿真了。ISE 6.1 支持在以下层次对电路进行仿真：

- 期望的仿真结果：逻辑验证。
- 行为仿真：逻辑和时序验证。
- 后仿真：布局步线后的逻辑和时序仿真
- 在工程文件窗口（Sources in Project window）中，选择测试激励文件（如 flipflop_tbw.tbw），可以看到处理流程窗口（Processes for Source window）中的 ModelSim Simulator 下面的仿真选项（见图 B.14）。
- 双击 Generate Expected Simulation Results，将运行一个后台逻辑仿真器来计算输出信号并自动启动包含了计算结果的 HDL Bencher，如图 B.15 所示。检查工程运行是否正确，然后不保存波形文件并退出 HDL Bencher。
- 双击 Simulate Post-Place & Route VHDL Model 启动 ModelSim，则会出现一个更细致的仿真器窗口。最大化波形窗口，选择 Zoom→Zoom Full，再次检查输出结果。

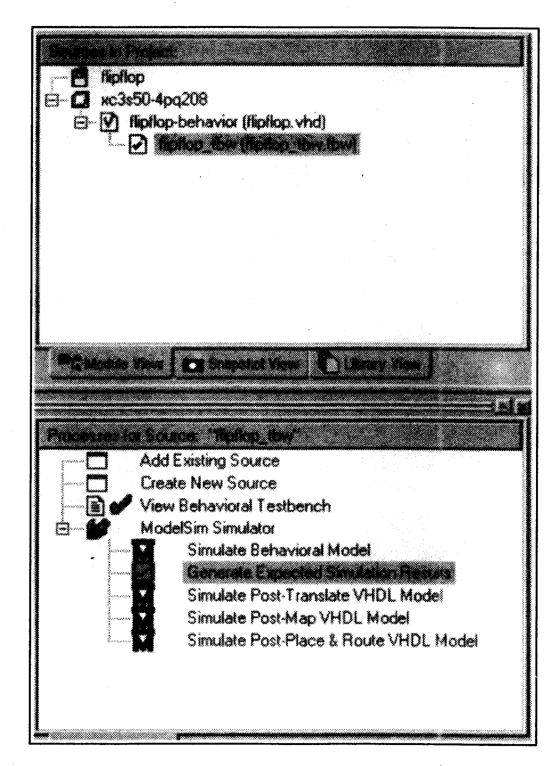


图 B.14

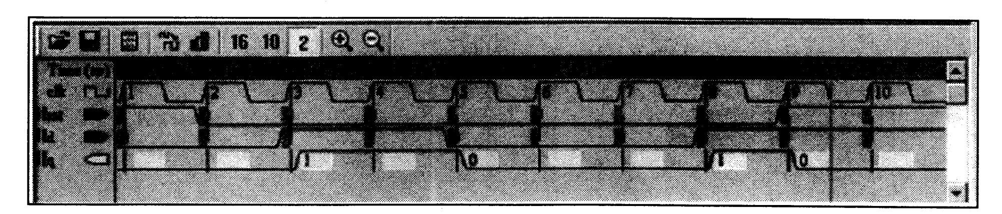


图 B.15

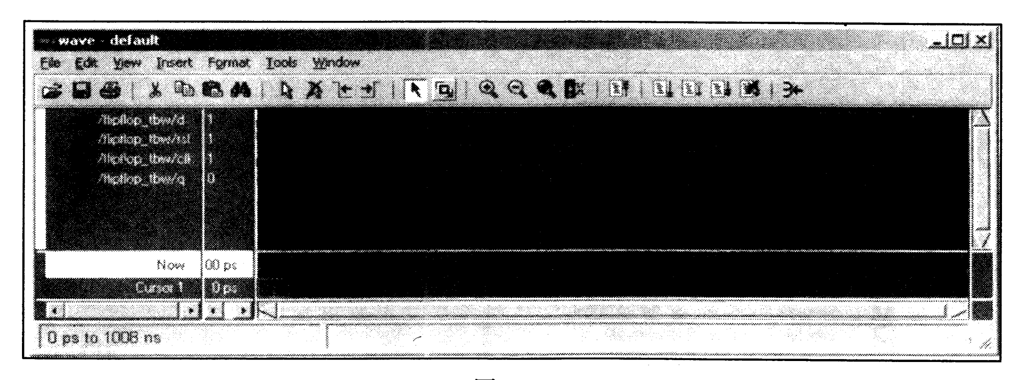


图 B.16

B.5 物理实现

为了使设计能够在 CPLD 或 FPGA 芯片上物理实现，还需要使用专门的开发套件或自己设计开发板。Xilinx 公司的大学推广计划会以低价向教育机构提供这些开发板。例如，Xilinx Didilab XC2 就是针对 Xilinx CoolRunner II 系列器件而设计的开发套件。开发板通过编程电缆与计算机相连，在 ISE 6.1 中进行相应的操作就可以实现对芯片的编程。

由于不同厂商提供的芯片编程方式是类似的，所以只在附录 C 和附录 D 中对这一过程进行了详细描述。

附录 C　Altera MaxPlus II 和 Advanced Synthesis Software 使用指南

Altera 提供的 MaxPlus II 是一个简单且界面友好的综合与仿真工具，其主要缺点在于对 VHDL 的支持不够全面，除少数相对简单的语法以外，大多需要借助外部综合软件（如 Leonardo Spectrum 或 Advanced Synthesis Software）进行综合。然而，正因为其简单性，所以很适合 VHDL 的初学者使用。另外，Altera 近期发布了支持大多数 VHDL 语法结构的 Advanced Synthesis Software（免费软件），这使 MaxPlus II 的功能更加强大。MaxPlus II 可以用于综合 VHDL 代码、生成可以进行设计实现和仿真的 EDIF 文件（.edf）。

读者可以从 www.altera.com 免费下载 MaxPlus II 10.2 Baseline 和 Advanced Synthesis Software 工具。

本文是一个简明指南，分为 5 部分：

- VHDL 代码输入
- 编辑
- 仿真
- 使用 Advanced Synthesis Software 进行综合
- 物理实现

C.1　VHDL 代码输入

- 启动 MaxPlus II 10.2 Baseline。
- 打开文本编辑器（MaxPlus II→Text Editor），或打开一个现有文件（File→Open），这样将显示一个图 C.1 所示的窗口。
- 输入 VHDL 代码（图 C.1 中显示的是一个 D 触发器），以扩展名.vhd 保存（本例中是 flipflop.vhd），注意在实体中也要采用相同的名称。

C.2　编译

- 针对当前文件建立工程：File→Project→Set Project to Current File。
- 选择目标器件（Assign→Device）。将出现图 C.2 中显示的下拉菜单，选择需要的器件（如 Family = MAX3000A，Device = AUTO）。

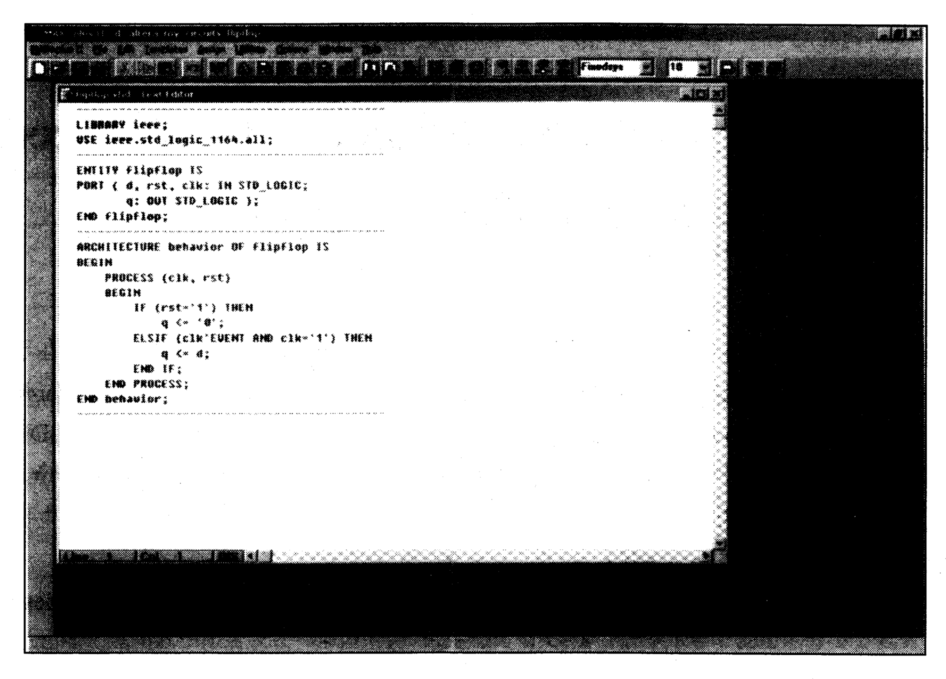


图 C.1

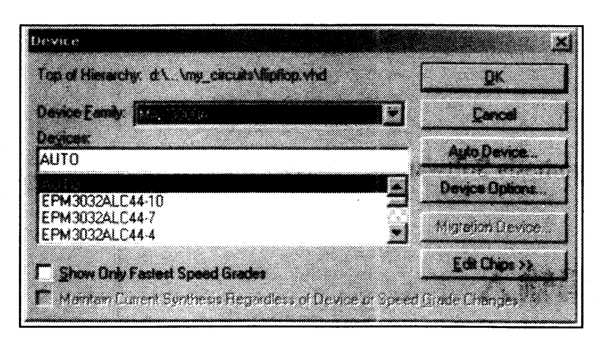


图 C.2

- 设置优化器，可以对速度或面积进行优化。选择 Assign→Global Project Logic Synthesis 将指针移到最左端（value = 0），代表对面积优化，将指针移到最右端（value = 10），代表对速度优化，value 值也可以在两者之间。
- 单击编译图标 ，然后单击 Start，以运行编译过程。
- 如果没有错误，将显示图 C.3 所示的窗口，它的上半部分显示了编译过程中生成的文件（注意与报告文件生成相关的 rpt 图标），下半部分显示了与当前处理流程相关的信息。
- 打开报告（.rpt）文件（双击它的图标，如图 C.3 所示）。验证引脚对应关系以及芯片中逻辑门和触发器的使用数量。图 C.1 所示电路的设计报告中的一部分如图 C.4 所示。

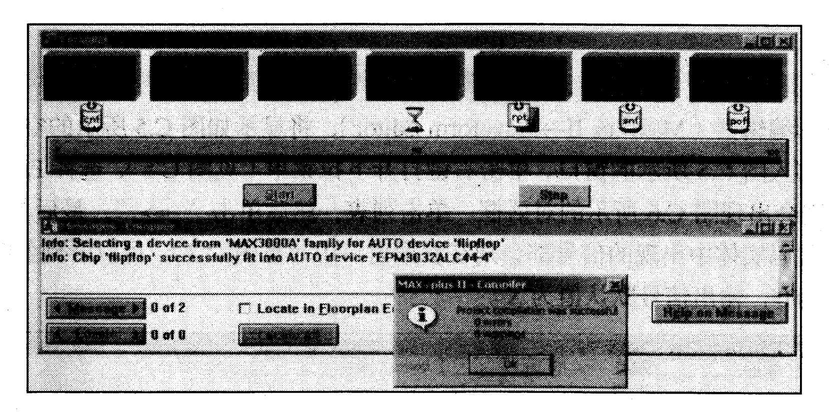


图 C.3

```
                  R                                       R
                  E                                       E
                  S        V                              S
                  E        C                              E
                  R        C                              R
                  V  r     I  G  G  G  c  G               V
                  E  s     N  N  N  N  l  N               E
                  D  t  d  T  D  D  D  k  D  q            D
               -----------------------------------------------_
               /  6  5  4  3  2  1 44 43 42 41 40             |
         #TDI |  7                                       39 | RESERVED
     RESERVED |  8                                       38 | #TDO
     RESERVED |  9                                       37 | RESERVED
          GND | 10                                       36 | GND
     RESERVED | 11                                       35 | VCCIO
     RESERVED | 12         EPM3032ALC44-4                34 | RESERVED
         #TMS | 13                                       33 | RESERVED
     RESERVED | 14                                       32 | #TCK
        VCCIO | 15                                       31 | RESERVED
     RESERVED | 16                                       30 | GND
          GND | 17                                       29 | RESERVED
              |_ 18 19 20 21 22 23 24 25 26 27 28      _|
               -----------------------------------------------
                  R  R  R  R  G  V  R  R  R  R  R
                  E  E  E  E  N  C  E  E  E  E  E
                  S  S  S  S  D  C  S  S  S  S  S
                  E  E  E  E     I  E  E  E  E  E
                  R  R  R  R     N  R  R  R  R  R
                  V  V  V  V     T  V  V  V  V  V
                  E  E  E  E        E  E  E  E  E
                  D  D  D  D        D  D  D  D  D


Total bidirectional pins required:                 0
Total reserved pins required                       4
Total logic cells required:                        1
Total flipflops required:                          1
Total product terms required:                      2
Total logic cells lending parallel expanders:      0
Total shareable expanders in database:             0

      Synthesized logic cells: 0/32    (0%)
```

图 C.4

C.3 仿真

- 打开波形编辑器（MaxPlus II→Waveform Editor），将显示如图 C.5 所示的空白窗口。
- 将指针移入图 C.5 所示的窗口，单击右键打开下拉菜单（见图 C.5），选择 Enter Nodes from SNF，则会出现图 C.6 所示的对话框。单击列表，然后单击“ => ”，最后单击 OK。所有 VHDL 代码实体中出现的信号都会列在波形窗口中（见图 C.7）。可以看到此时输入信号的默认值为'0'，输出信号默认值为'X'。

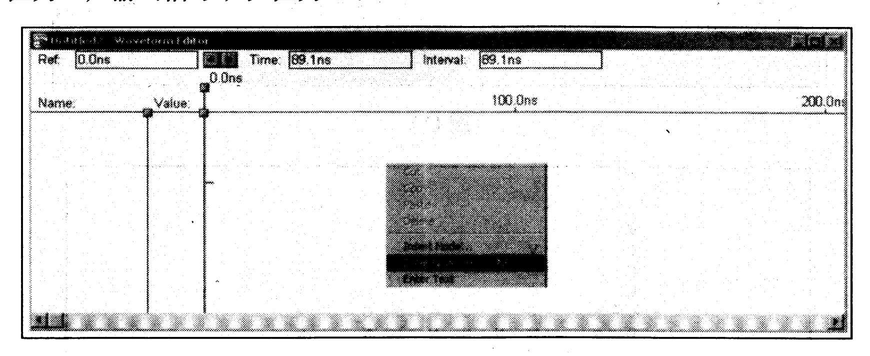


图 C.5

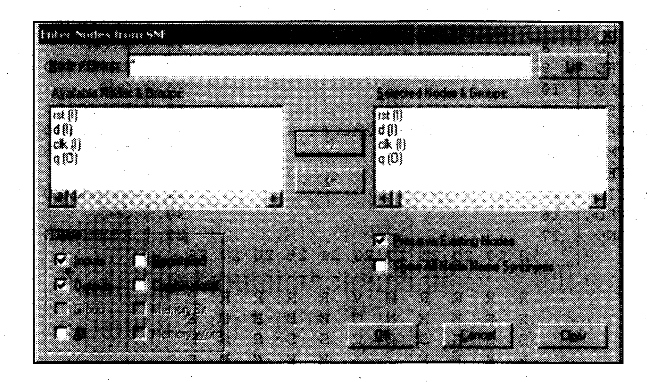


图 C.6

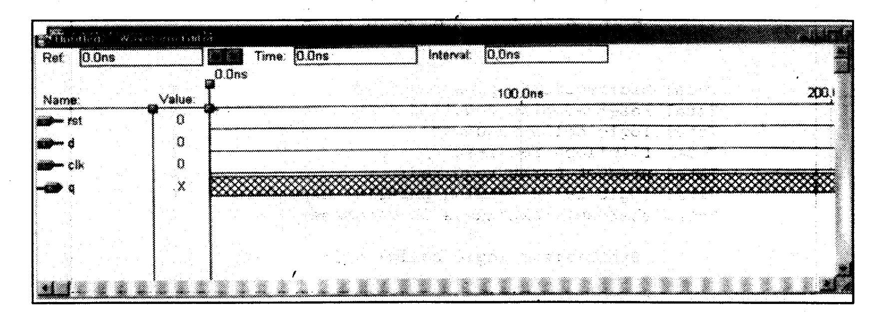


图 C.7

- 在设定信号值之前，要先设定波形长度和网格大小。选择 File→End Time，输入 1 μs 来设定长度，选择 Options→Grid Size，输入 50 ns 来设定网格大小。最后，选择 View→Fit in Window。可以通过拖动信号来改变其排列顺序。例如，要把 clk 作为第一个信号，只需将指针移到 clk 前的箭头上，按住左键不放拖到所需位置即可（如图 C.8 所示）。

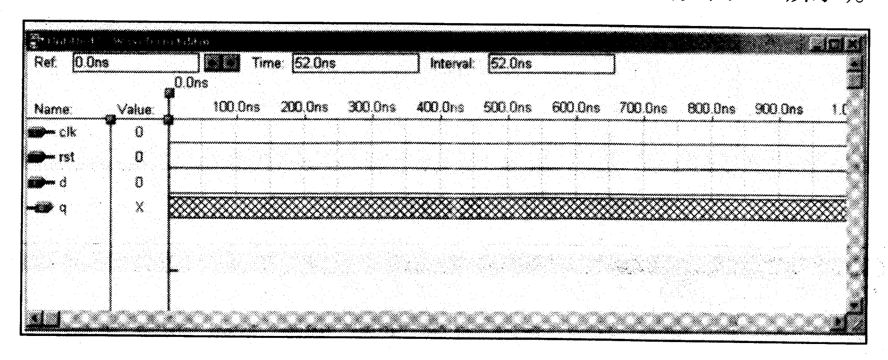
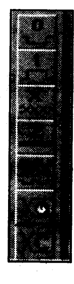


图 C.8

- 现在通过使用图 C.9 所示的工具定义输入信号。选择适当的图标，可以设置时钟信号，逻辑'0'、逻辑'1'、计数器值以及总线的数值。
- 时钟设置。选择对应的信号线（在 clk 上单击左键），然后单击脉冲生成图标（见图 C.9），将出现图 C.10 所示的对话框。设置时钟初始值（Starting Value）为 0 和倍频值 Multiplied By 为 1。这里的 Multiplied By 值取'1'表示时钟周期为两个时隙，每个时隙的宽度为 1 个网格（此处为前面设定的 100 ns）。



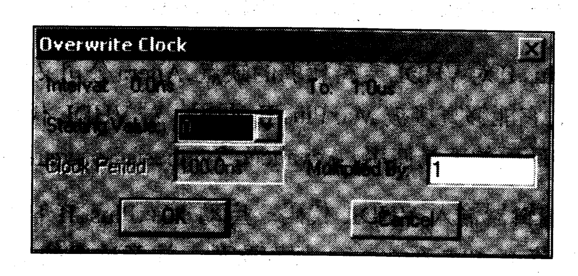




图 C.9　　图 C.10

- 设置其他输入信号。对于 rst 信号，选择前两个时隙（0~100 ns），单击逻辑 1 设置图标，设定其在这一时间间隔内的值为 1，然后选择整个信号 d，单击脉冲生成图标，输入 Multiplied By 为 4，单击 OK，得到图 C.11 所示的波形。
- 以后缀名.scf 来保存波形文件。
- 现在可以对设计进行仿真了，单击图标来启动仿真器，仿真器自动对波形编辑器中的所有输出信号的值进行填充，如图 C.12 所示。

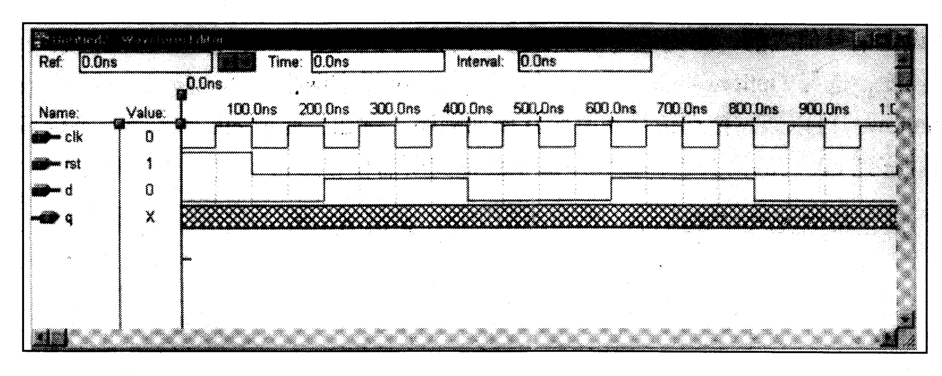


图 C.11

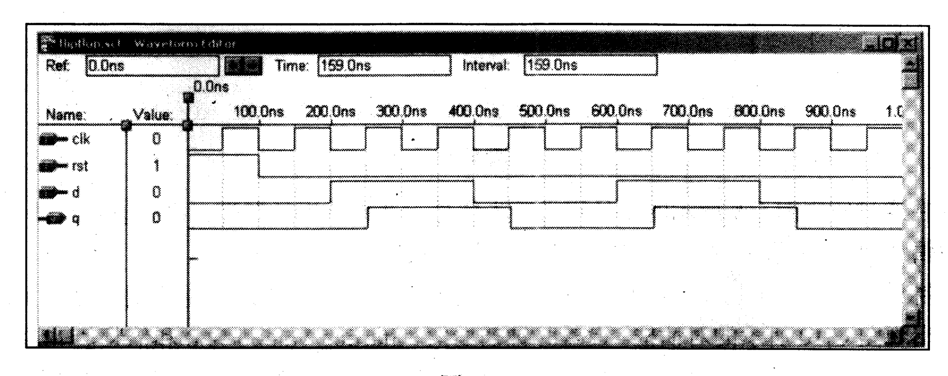


图 C.12

C.4　使用 Advanced Synthesis Software 进行综合

为克服 MaxPlus II 对 VHDL 支持较弱的缺点，Altera 公司又发布了 Advanced Synthesis Software 以弥补这一缺陷，它能够直接被 MaxPlus II 调用，对 VHDL 代码进行综合并生成 EDIF（.edf）文件，供设计流程中后面的工具使用。Advanced Synthesis Software 可以从 www.altera.com 免费下载。

- 使用文本编辑器输入 VHDL 代码。启动 MaxPlus II 自带的文本编辑器输入 VHDL 代码（见 C.1 节），以扩展名.vhd 保存文件，注意这里的文件名与实体名相同（在本例中是 flipflop.vhd）。
- 启动 Advanced Synthesis Software，则会出现图 C.13 所示的窗口。
- 首先创建一个新的工程。选择 File→New Project，在对话框中输入工程名称（与实体相同），该工程以扩展名.max2syn 保存（在本例中是 flipflop.max2syn）。
- 接着在工程中添加 VHDL 文件。选择 Assign→Add/remove HDL files，则会出现图 C.14 所示的对话框，选择 Add，选择要添加的文件并单击 OK。
- 单击综合参数设置图标，则会出现图 C.15 所示的对话框，选择目标器件（如 MAX3000A）和 VHDL 93 标准。

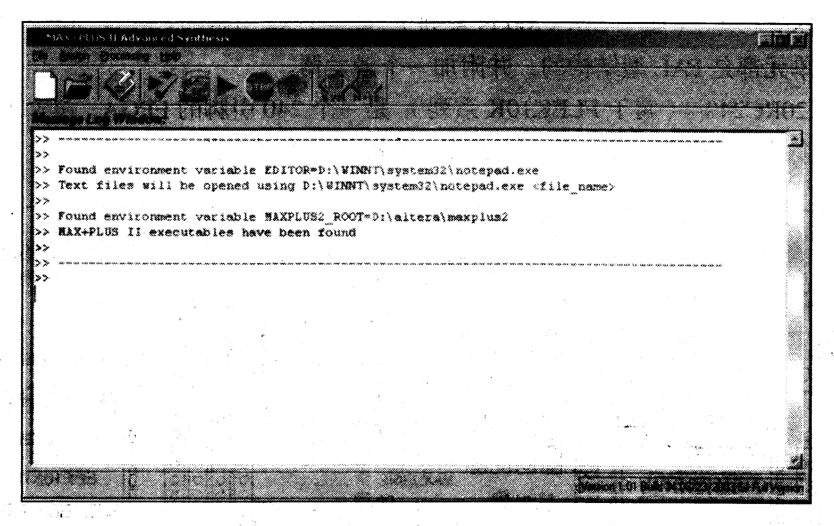


图 C.13

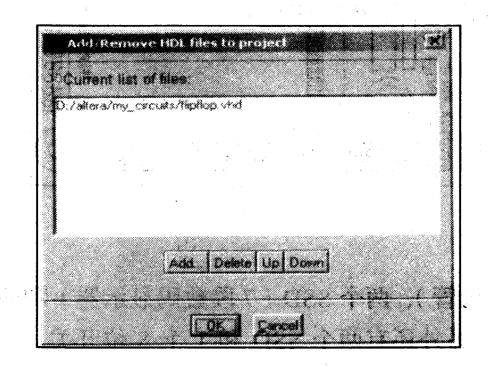


图 C.14

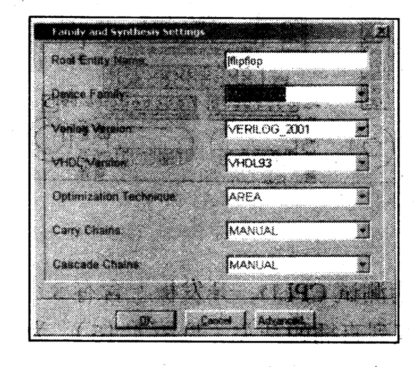


图 C.15

- 单击综合图标 。如果没有综合错误，就会生成一个扩展名为.edf 并与工程名同名的 EDIF 文件（在本例中是 flipflop.edf）。
- 启动 MaxPlus II 并调用 Advanced Synthesis Software 生成的 EDIF 文件（File→Open），此后进行 C.2 节中所述的操作步骤来编译新的工程。

C.5　物理实现

这里将讨论采用 CPLD 在物理上实现所设计电路的步骤。下面将介绍 Altera 公司的大学推广计划中所用的名为 UP1 的 CPLD 开发板。当然也可以使用其他开发板。实际上，大多数 CPLD/FPGA 制造商都提供了低价的开发板作为其大学计划中的一部分。

Altera 的 UP1 开发板

Altera 的 UP1 开发板如图 C.16 所示，包含两块芯片：

- EPM7128SLC84-7（属于 MAX7000S 系列）：是一个 84 引脚的 CPLD，包含 128 个宏单元，每个宏单元都是 PAL 型结构的，并附加一个触发器。
- EPF10K20RC240-4（属于 FLEK10K 系列）：是一个 240 引脚的 FPGA，包含 1152 个逻辑单元，每个逻辑门含有一个 4 输入查找表，并附加一个触发器。

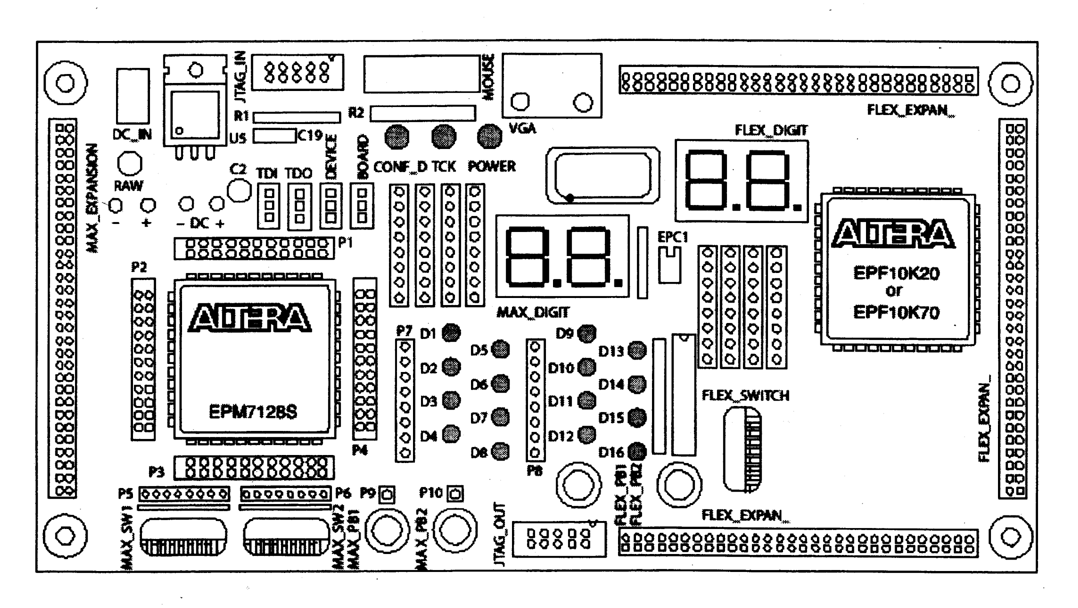


图 C.16

为测试 CPLD，开发板上含有 8 个 LED（发光二极管）、两个 SSD（7 段数码显示器）和两个 8 位双列直插的开关（见图 C.16）。为了测试 FPGA，另外还有两个 SSD 和 1 个 8 位开关。LED 和 SSD 的每一段都是低电平有效的，而开关向上拨动时提供高电平 5 V，向下拨动时提供低电平 0 V。

LED 和开关并没有连接到引脚上，在设计中可以根据各种具体开发要求连接到器件上。然而，SSD 已经连接到器件上，也就有了特定的引脚设定。在测试时，SSD 的引脚连接如图 C.17 所示。

Table 4. MAX_DIGIT Segment I/O Connections

Display Segment	Pin for Digit 1	Pin for Digit 2
a	58	69
b	60	70
c	61	73
d	63	74
e	64	76
f	65	75
g	67	77
Decimal point	68	79

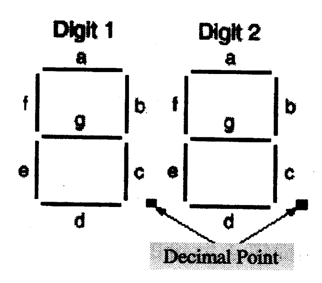


图 C.17

开发板上有一个 25.175 MHz 的时钟，它已连接到器件的全局时钟引脚上（CPLD 的全局时钟引脚是第 83 个引脚）。

关于该开发板的详细描述可参见 www.altera.com/literature/univ/upds.pdf。

设定 UP1 开发板

- 在下面的介绍中将使用 CPLD(EPM7128SLC84-7)作为目标器件，因此 TDO，TDI，DEVICE 和 BOARD 栏中的跳线都应该跨接在 C1 和 C2 上（即图 C.18 中箭头所指示的一行）。

Table2. JTAG Jumper Settings

Desired	TDI	TDO	DEVICE	BOARD
→ Program EPM7128S device only	C1&C2	C1&C2	C1&C2	C1&C2
Configure FLEX10k device only	C2&C3	C2&C3	C1&C2	C1&C2
Program/configure both devices	C2&C3	C1&C2	C2&C3	C1&C2
Connect multiple boards together	C2&C3	OPEN	C2&C3	C2&C3

图 C.18

- 将 ByteBlaster 编程电缆连接在开发板和 PC 机之间。
- 接通开发板的直流电源（9 V），此时电源 LED 和两个 SSD 均有显示。

设计实现

如果已经启动了 MaxPlus II 10.2 Baseline 且 VHDL 代码已经输入并编译完毕，下面将继续本附录前面的工作。

- 指定目标器件。选择 Assign→Device 并设定 Family = MAX7000S 和 Device = EPM 7128SLC84-7（不要选择 Select Only Fastest Speed Grade 选项）。
- 进行电路编译。
- 打开报告文件(.rpt)并检查各个信号和引脚的对应关系。如果无需改变，进入“设计下载”一节，否则按下面的说明进行操作。
- 选择一个不同于系统自动分配的时钟引脚（引脚 83）：选择 Assign→Global Project Logic Synthesis，不选 Clock under Automatic Global。
- 为了进行引脚分配：选择 Assign→Pin/Location/Chip→Search→List，这样会出现图 C.19 左边所示的窗口。选好信号和时钟后单击 OK，则会出现图 C.19 右边的窗口。选择引脚号和引脚类型（输入和输出等），单击 OK。如果有多个信号，则重复上面的步骤来指定引脚。
- 返回 MaxPlus II 主窗口，重新编译。检查报告文件，确定是否按要求实现了引脚对应关系。

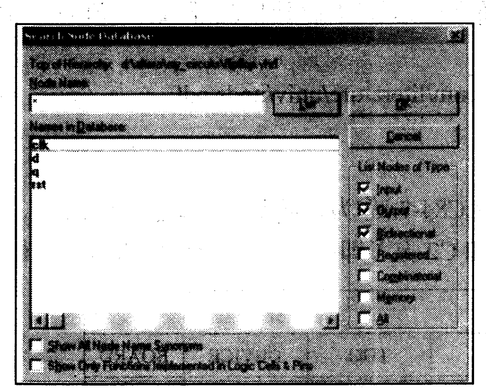
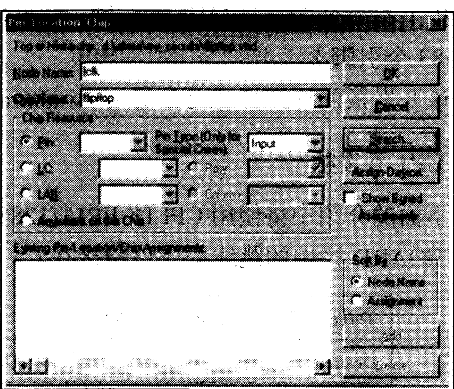



图 C.19

设计下载

现在将设计下载到芯片上。

- 双击文件编译过程结束时出现的 pof（工程对象文件）图标（见图 C.3），则会出现图 C.20 所示的窗口。
- 在主菜单中，选择 Options→Hardware Setup→ByteBlaster(MV)，然后单击 OK。
- 最后，在图 C.20 所示的窗口中单击 Program 进行器件程序下载，然后该器件就能进行物理测试或应用了。

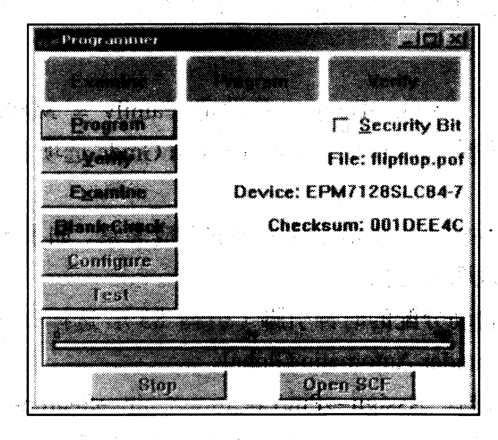


图 C.20

附录 D　Altera Quartus II 使用指南

Quartus II 3.0 是 Altera 公司提供的一套集成了编译、布局布线和仿真工具在内的综合开发环境。它能完成从代码输入到物理实现的全部设计流程，支持 Altera 公司的所有 FPGA 和 CPLD 器件，是 MaxPlus II（见附录 C）的后继版本。

从 www.altera.com 可以免费下载 Quartus II 3.0 开发环境。

本文是一个简明的工具应用指南，分为四部分：

D.1　VHDL 代码输入

D.2　编译

D.3　仿真

D.4　物理实现

D.1　VHDL 代码输入

- 启动 Quartus II 3.0，则会显示图 D.1 所示的界面。

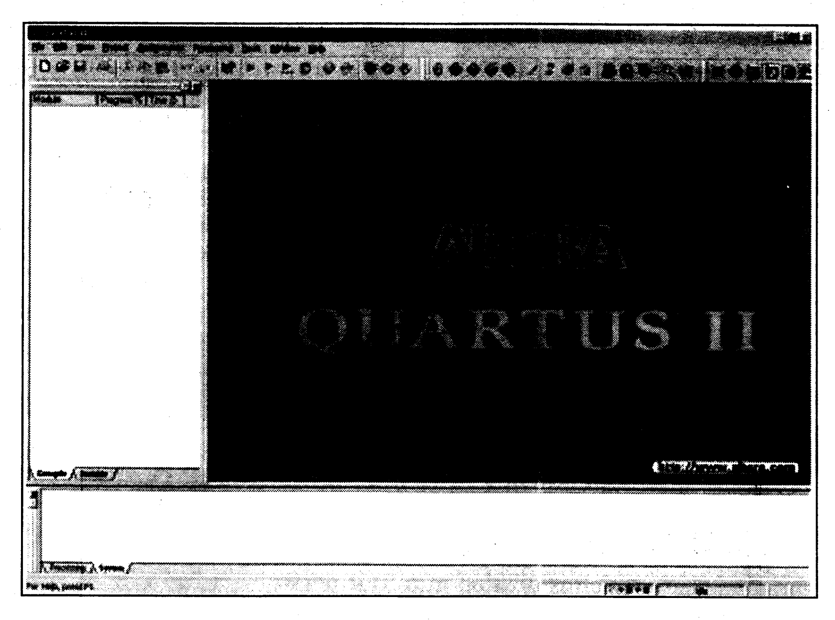


图 D.1

- 首先新建一个工程，选择 File→New Project Wizard，则会显示图 D.2 所示的对话框，在第一栏中输入工作路径，在第二栏中输入工程名称（应与实体名称相同），此时最后一栏中系统将自动填入与工程相同的名称（可以根据需要进行修改）。在下面的例子中，工作路径为 d:\altera\my_circuits，工程名称为 flipflop。这样，工作路径中就创建了一个名为 flipflop.quartus 的新工程，同时包含新生成的 flipflop.vhd 文件。
- 打开文本编辑器（选择 File→New，或单击相关 ▣），则会出现图 D.3 所示的菜单。选择 VHDL File 后会出现一个空白窗口。
- 输入 VHDL 代码（如图 D.4 所示），以后缀名.vhd 保存（该文件的实体也被会自动指定为相同的名称，在本例中是 flipflop.vhd）。
- 检查语法错误。选择 Processing→Analyze Current File 或单击分析图标 ▣。编译器所发现的错误都将显示在底部的窗口中。

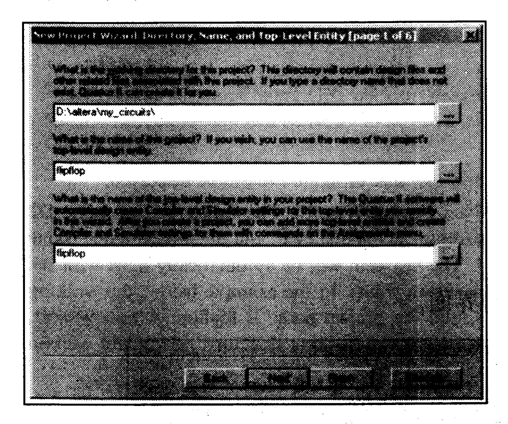


图 D.2

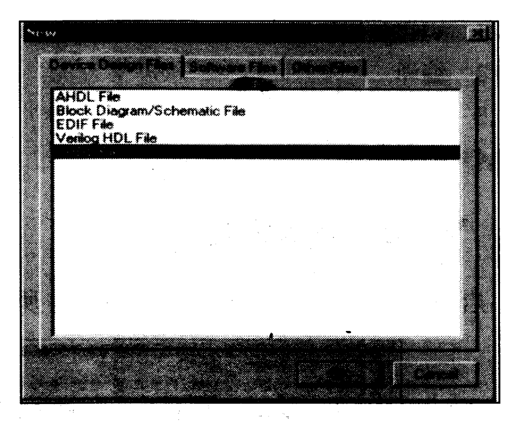


图 D.3

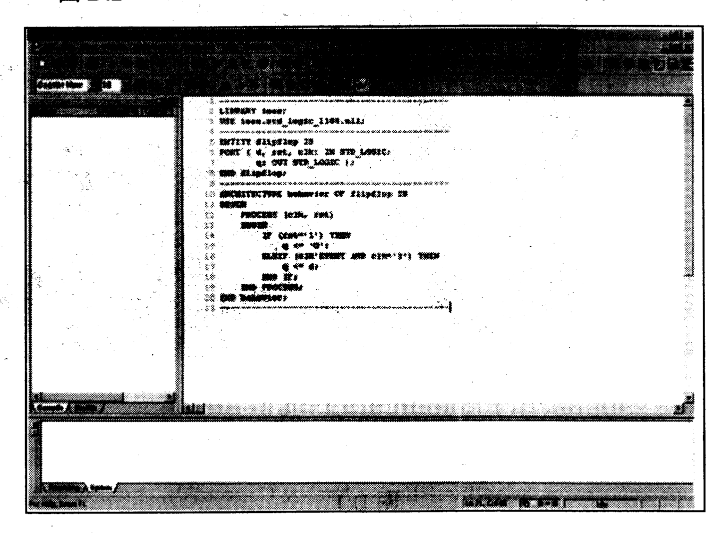


图 D.4

D2　编译

- 选择目标器件（Assignments→Devices），则会出现图 D.5 所示的菜单。选择所需的器件系列（如 MAX3000A）。在 Target device 选项中，可以选 Auto device。至于 Package，Pin count 和 Speed grade 选项，则可以任选一个。

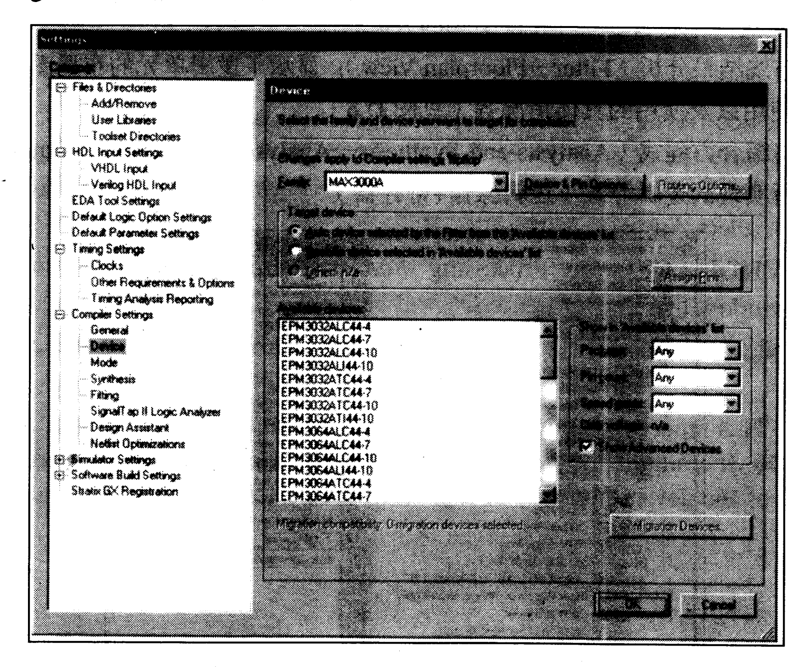


图 D.5

- 选择 Processing→Start Compilation 或单击图标 ▶。如果没有错误，则显示图 D.6 所示的窗口。
- 检查编译报告（如图 D.6 中左半部分所示），至少要做下列检查：

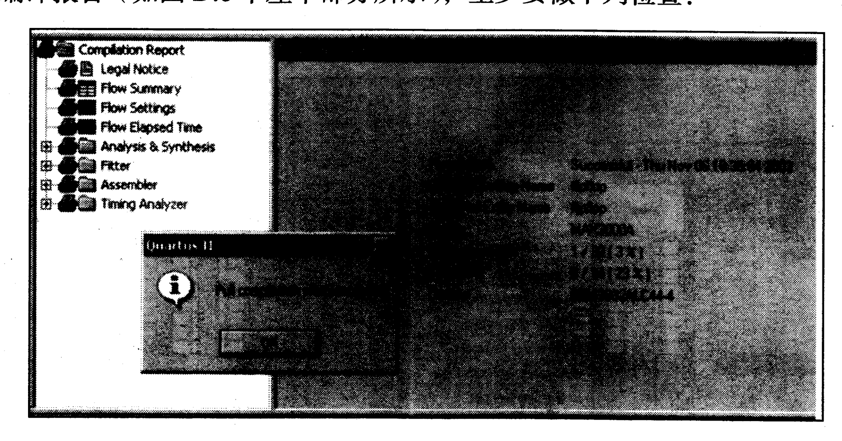


图 D.6

（a）Flow Summary：如图 D.6 所示，在编译过程结束时会自动出现该报告，其中包含器件名称、引脚使用数量和器件资源利用率（逻辑单元使用个数/逻辑单元总数）。

（b）资源使用情况汇总（Fitter→Resource Section→Resource Usage Summary）：该报告显示了整个设计使用的寄存器数量、逻辑单元数量和 I/O 引脚数量。

（c）输入和输出引脚（Fitter→Resource Section→Input Pins，Fitter→Resource Section→Output Pins）：这两个报告显示了 I/O 引脚的分配情况（见图 D.7）。

（d）布局布线结果分析（Fitter→Floorplan View）：显示了逻辑单元的布局图，包括哪些逻辑门被使用以及使用的具体情况等（见图 D.8）。

（e）分析与综合方程式（Analysis and Synthesis→Analysis and Synthesis Equations）：包括编译器所实现的逻辑等式（逻辑操作+寄存器）。

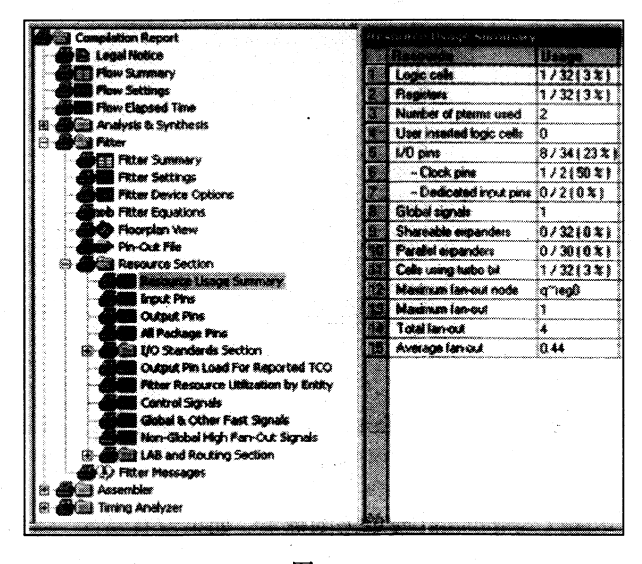


图 D.7

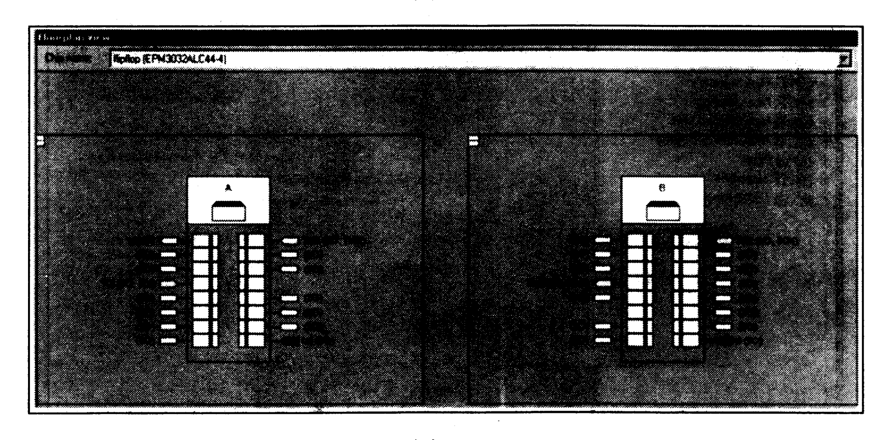


图 D.8

D.3　仿真

- 打开波形编辑器（File→New→Other File→Vector Waveform File 或单击对应图标），则会出现图 D.9 所示的窗口。
- 为定义波形大小，应进行以下操作（见图 D.9）：

  Edit→End Time（如 500 ns）

  Edit→Grid Size（设定 Period = 50 ns，Duty Cycle = 50%）

  最后，选择 View→Fit in Window

  注意，可以通过 Tools→Options→Waveform Editor→General 来改变设定的默认值。
- 把输入和输出信号添加到波形窗口。在 Name 下的空白区域中单击右键（见图 D.9）选择 Insert Node or Bus。在下一个窗口中，选择 Node Finder，则会出现图 D.10 所示的窗口。将 Filter 设为 Pins: all。单击 Start，然后单击“>>”，最后单击 OK，波形窗口中就会显示出 VHDL 代码中实体所包含的全部信号（见图 D.11）。注意输入信号用一个标有“I”的向内的箭头来标记，而输出信号用一个标有“O”的向外的箭头来标记。信号的位置可以通过拖动来改变。

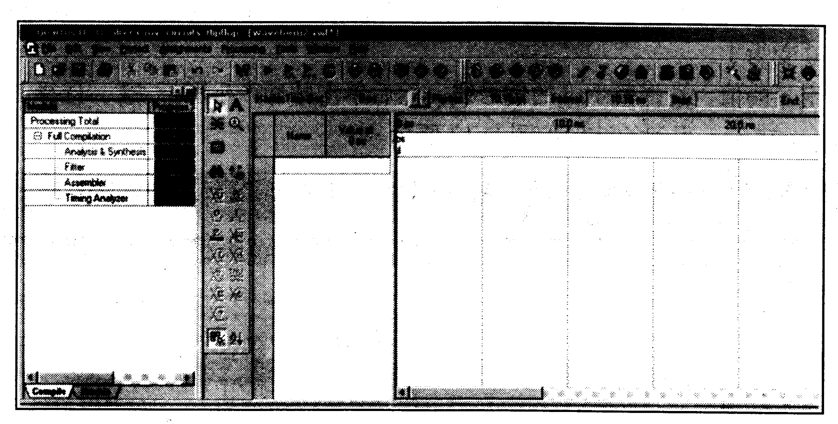


图 D.9

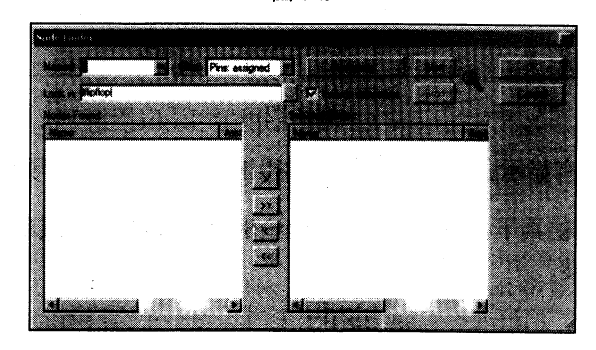


图 D.10

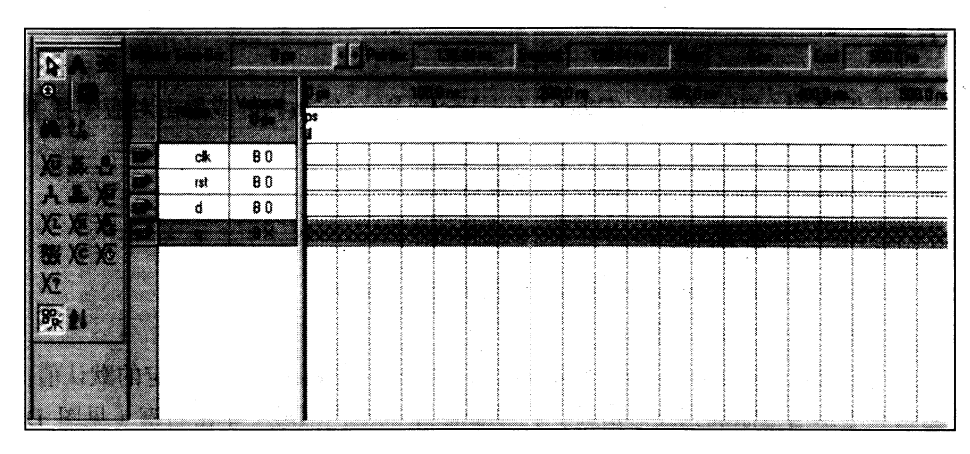


图 D.11

- 设定输入信号的值（图 D.11 中的 clk，rst 和 d）：最简单的方法是使用波形菜单（如图 D.11 中的最左部所示）。为了设定时钟信号，可以选择整个时钟信号线，然后单击脉冲产生图标 ，就会出现一个设定窗口，设定 Period = 100 ns。
- 对于 rst 信号，选择它的一段起始时间（0~25 ns），然后单击图标 ，在所选位置上将其逻辑值由 0 变为 1。
- 最后设定 d 的值。选择整个 d 的信号线，然后单击脉冲产生图标 ，选择 Period = 200 ns 和 Phase = 75 ns，得到的波形如图 D.12 所示。

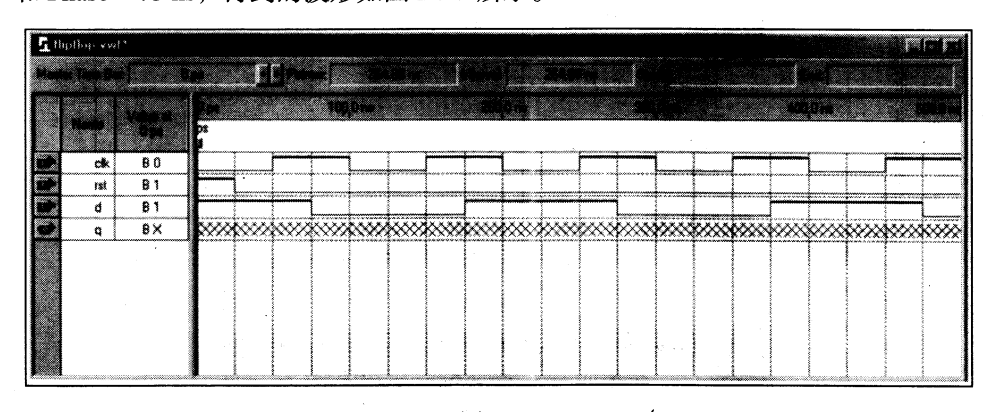


图 D.12

注意，q 的值并没有显示，它的值由仿真器得出，波形文件保存为 flipflop.vwf。

- 现在系统可以进行仿真了。选择 Processing→Start Simulation 或单击相应图标 ，将得到图 D.13 所示的结果。

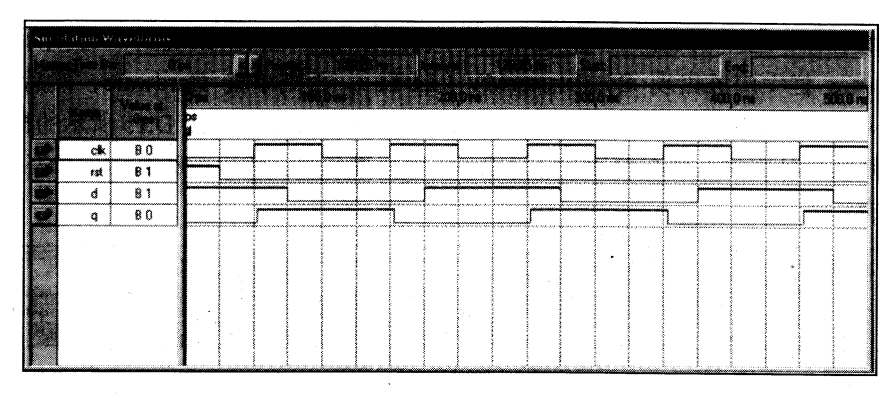


图 D.13

D.4　物理实现

- 开发工具箱：为了进行电路的物理实现，假设采用 Altera UP1 或 UP2 开发板（见附录 C 的 C.5 节），则必须将开发板提供的编程电缆 ByteBlaster 连接到 PC 机的并行口上。
- 器件选择：工具箱（Altera UP1 或 UP2）上包含两个器件，EPM7128SLC84-7（属于 MAX7000S 系列的一种 CPLD）和 EPF10K70RC240-4（属于 FLEK10K 系列的一种 FPGA）。在 D.2 节中选择 Assignments→Devices 时，必须选择这两种器件的其中一种。
- 改变引脚对应关系：在编译过程中，I/O 引脚的分配是有系统默认的。如果需要改变，选择 Assignments→Assign Pins，则会出现图 D.14 所示的窗口。假设需要将 rst 分配到第 4 个引脚，则选择第 4 个引脚，单击图标，打开图 D.10 所示的窗口。单击 Start，在左边一栏中选中 rst，然后单击“>”，最后单击 OK。回到图 D.14 所示的窗口后，单击 Add。其他引脚对应关系的改变方式与此类似。

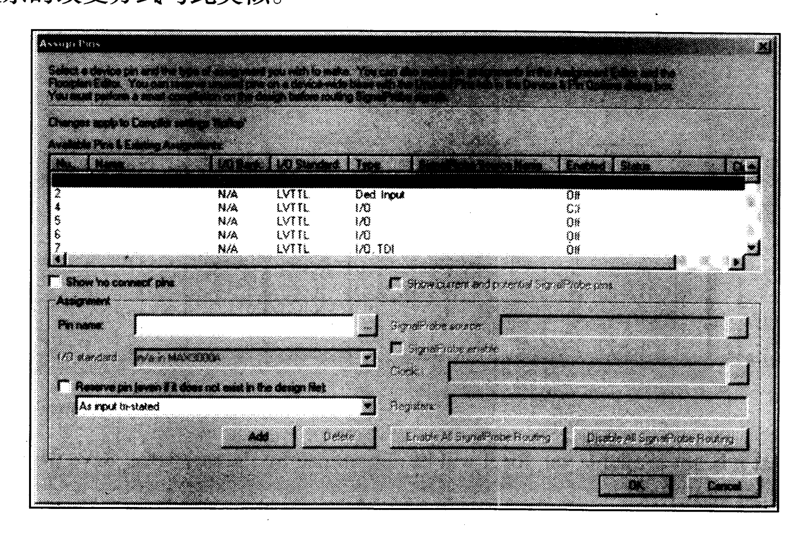


图 D.14

- 设定编程器：选择 Tools→Programmer 或单击图标，这时会出现图 D.15 所示的窗口。在 Hardware 栏中应该出现 ByteBlasterMV（LPT1），否则单击 Hardware，然后单击 Select Hardware，选择 ByteBlasterMV，最后单击 Add Hardware，回到图 D.15 所示的窗口。在 File 栏中核实设计文件，现在设计文件是以.pof 为后缀名的。然后查看 Program/Configure 下的对话框。

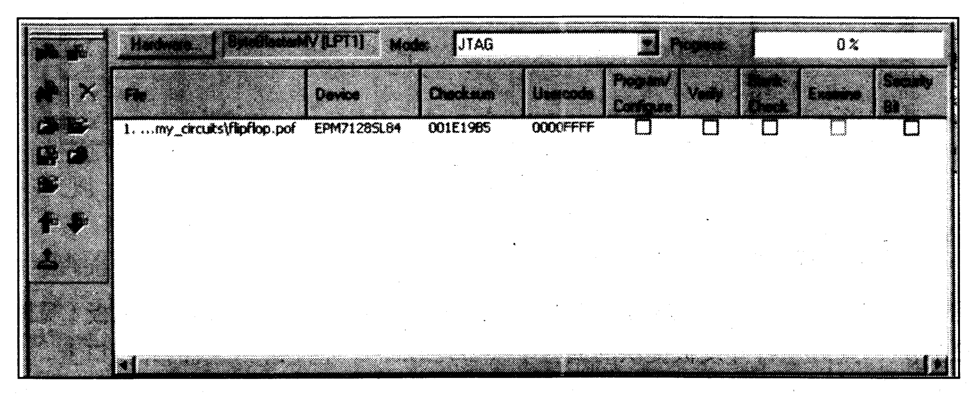


图 D.15

- 对器件进行编程：最后对器件进行编程。选择 Processing→Start Programming，不久编程就可以完成，该器件就能进行物理测试或应用了。

VHDL 保留字

From VHDL 87:	ENTITY	OPEN	WAIT
	EXIT	OR	WHEN
ABS	FILE	OTHERS	WHILE
ACCESS	FOR	OUT	WITH
AFTER	FUNCTION	PACKAGE	XOR
ALIAS	GENERATE	PORT	
ALL	GENERIC	PROCEDURE	From VHDL 93:
AND	GUARDED	PROCESS	
ARCHITECTURE	IF	RANGE	GROUP
ARRAY	IN	RECORD	IMPURE
ASSERT	INOUT	REGISTER	INERTIAL
ATTRIBUTE	IS	REM	LITERAL
BEGIN	LABEL	REPORT	POSTPONED
BLOCK	LIBRARY	RETURN	PURE
BODY	LINKAGE	SELECT	REJECT
BUFFER	LOOP	SEVERITY	ROL
BUS	MAP	SIGNAL	ROR
CASE	MOD	SUBTYPE	SHARED
COMPONENT	NAND	THEN	SLA
CONFIGURATION	NEW	TO	SLL
CONSTANT	NEXT	TRANSPORT	SRA
DISCONNECT	NOR	TYPE	SRL
DOWNTO	NOT	UNITS	UNAFFECTED
ELSE	NULL	UNTIL	XNOR
ELSIF	OF	USE	
END	ON	VARIABLE	

参考文献

Armstrong J. R. and F. G. Gray, *VHDL Design Representation and Synthesis*, Englewood Cliffs, NJ: Prentice Hall, 2nd Edition, 2000.

Bhasker J., *VHDL Primer*, Englewood Cliffs, NJ: Prentice Hall, 3rd Edition, 1999.

Chang K. C., *Digital Systems Design with VHDL and Synthesis—An Integrated Approach*, Los Alamitos, CA: IEEE Computer Society Press, 1999.

Hamblen J. and M. Furman, *Rapid Prototyping of Digital Systems*, Boston: Kluwer Academic Publisher, 2nd Edition, 2001.

Naylor D. and S. Jones, *VHDL: A Logic Synthesis Approach*, London: Chapman & Hall, 1997.

Navabi Z., *VHDL Analysis and Modeling of Digital Systems*, New York: McGraw-Hill, 1993.

Pellerin D. and D. Taylor, *VHDL Made Easy*, Englewood Cliffs, NJ: Prentice Hall, 1997.

Perry D. L., *VHDL*, New York: McGraw-Hill, 2nd Edition, 1994.

Yalamanchili S., *Introductory VHDL from Simulation to Synthesis*, Englewood Cliffs, NJ: Prentice Hall, 2001.

Yalamanchili S., *VHDL Starter's Guide*, Englewood Cliffs, NJ: Prentice Hall, 1998.